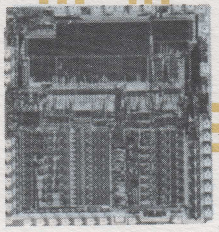
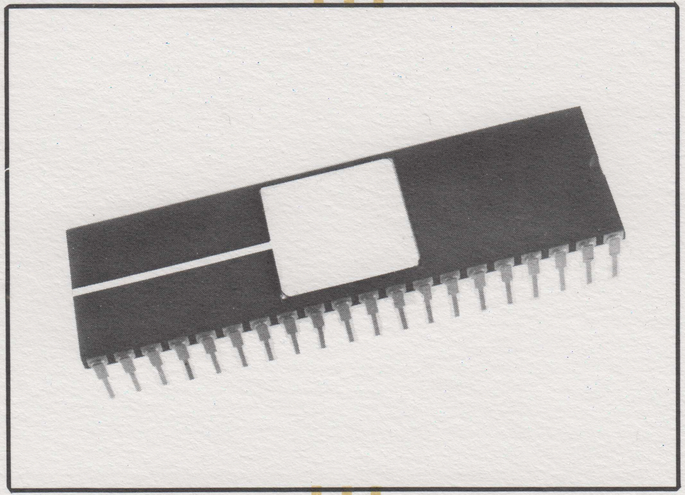
MAC-8

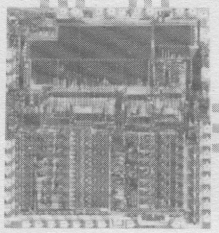


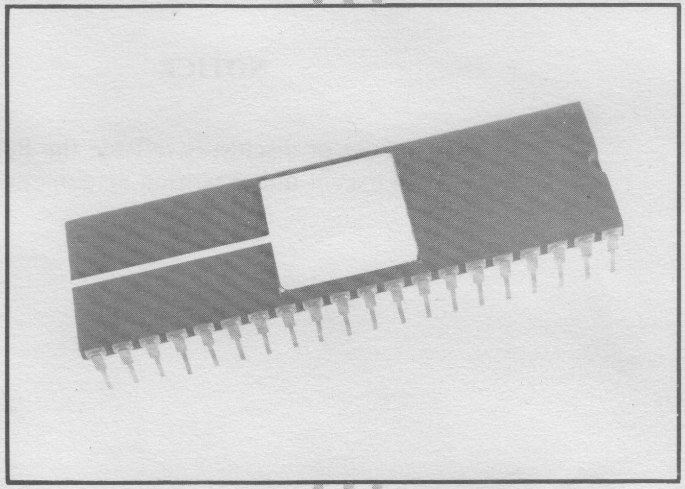


PROCESSOR
USER'S
MANUAL

Bell Laboratories

MAC-8





PROCESSOR
USER'S
MANUAL

Bell Laboratories

NOTICE

Not for use or disclosure outside the Bell System except under written agreement.

Prepared and published by
Technical Documentation Department
Bell Laboratories
Murray Hill, New Jersey
September 1978

MAC-8 PROCESSOR USER'S MANUAL

CONTENTS

1. INTRODUCTION 1-1
  1.1 Device Overview 1-1
    1.1.1 Microcomputer Operation 1-2
    1.1.2 Functional Elements 1-3
    1.1.3 Operation 1-6
  1.2 Software Development System 1-8
  1.3 Hardware Development Systems 1-11
2. ARCHITECTURE 2-1
  2.1 Pin Descriptions 2-1
  2.2 Registers 2-5
  2.3 Stack 2-7
  2.4 CPU Status 2-8
  2.5 Interrupt, Trap, Reset 2-10
  2.6 Timing 2-11
3. INSTRUCTION SET 3-1
  3.1 Programming 3-1
    3.1.1 Format 3-1
    3.1.2 Addressing Modes 3-2
    3.1.3 Conditions 3-4
  3.2 Description Notation 3-6
    3.2.1 Coded Instruction 3-6
    3.2.2 Operation 3-8
    3.2.3 Addressing 3-8
    3.2.4 Instruction Length 3-8
    3.2.5 Conditions 3-8
    3.2.6 Examples 3-9
  3.3 Arithmetic and Logical Instructions 3-9
    3.3.1 Monadic 8-bit Group 3-9
    3.3.2 Monadic 16-bit Group 3-16
    3.3.3 Dyadic 8-bit Group 3-18
    3.3.4 Dyadic 16-bit Instructions 3-23
  3.4 Control Transfer Instructions 3-27

3.4.1 Call Subroutine Instructions 3-27

3.4.2 Return Instructions 3-29

3.4.3 Branch Instructions 3-31

3.5 Special Register Instructions 3-37

3.5.1 Load Instructions 3-37

3.5.2 Condition Register Instructions 3-40

3.5.3 Register Pointer Instructions 3-41

3.5.4 Stack Instructions 3-43

3.5.5 Add 8-bits Instructions 3-48

3.6 Miscellaneous Instructions 3-49

4. SPECIFICATIONS 4-1

4.1 Operating Characteristics 4-1

4.1.1 Standard MAC-8 (2-MHz Operation) 4-2

4.1.2 Slower MAC-8 4-5

4.2 Input and Output Waveform 4-7

4.3 Package Dimensions 4-8

5. SYSTEM OPERATION 5-1

5.1 Interfacing 5-1

5.1.1 Access Timing 5-1

5.1.2 Wait State Generation 5-2

5.1.3 CLK IN Control 5-4

5.1.4 MAC-8 Status 5-5

5.1.5 Interrupt and Reset Timing 5-5

5.2 Peripherals 5-5

5.2.1 WE 28A RAM 5-7

5.2.2 WE 146D Peripheral Timer 5-9

5.2.3 Intel 8251/A Programmable Communication Interface 5-10

5.2.4 Intel 8214 Priority Interrupt Controller 5-12

5.2.5 Intel 8253 Programmable Interval Timer 5-13

5.3 System Integration 5-14

5.3.1 Automated Coin Toll Service (ACTS) 5-14

5.3.2 Programmed E-Telemetry (PROETEL) System 5-15

5.3.3 PLAID 5-16

5.3.4 MAC-8 MDS CPU Module 5-17

6. APPENDIX – REFERENCE TABLES 6-1

6.1 Assembly Language Tokens 6-1
  6.1.1 Keywords 6-1
  6.1.2 Identifiers 6-4
  6.1.3 Constants 6-4
  6.1.4 Operators 6-4
  6.1.5 Punctuation 6-6
  6.1.6 Blanks and Tabs 6-6
  6.1.7 Comments 6-7
6.2 Opcode Assignments 6-7
6.3 Addressing Modes 6-10
6.4 Instruction Length 6-11

Chapter 1

INTRODUCTION

1. INTRODUCTION

Semiconductor technology has advanced to the point where thousands of transistors can be placed on a single chip. The resulting devices, when suitably programmed, can rival the capabilities of many minicomputers while offering small size, low power requirements, and low manufacturing cost. Such devices become particularly attractive when they can satisfy different applications merely through the user's programming and choice of peripheral equipment.

The MAC-8 microprocessor uses such large-scale integration (LSI) techniques to provide a device suitable for a variety of communications equipment. It has a comprehensive instruction set for performing logical and integer arithmetic operations on 8- or 16- bit quantities. Programs can be written in either a high-level language, assembly language or hexadecimal code. An extensive collection of support software is available for the debugging and testing of programs for a variety of applications.

This manual, which is intended primarily for the hardware designer, emphasizes the system design aspects of the MAC-8 microprocessor. The first chapter contains an overview of microcomputers and their operation, but pays particular attention to the features of the MAC-8 microprocessor. It also briefly describes the MAC-8 software and the hardware development systems, such as the MAC Tutor and the Program Logic Aid (PLAID). Later chapters provide detailed information on the MAC-8's architecture, instruction set, and specifications. A final chapter discusses the areas of system design affected by the MAC-8's operating characteristics.

More complete information on programming the MAC-8 and using its support facilities is contained in the *MAC-8 Programmer's Manual* and the MAC Tutor[1] and PLAID[2,3] documents.

1.1 Device Overview

The MAC-8 is implemented in silicon-gate complementary metal-oxide semiconductor (CMOS) technology, which allows both p- and n-channel transistors to appear on the same chip. CMOS offers greater flexibility in circuit design than PMOS (p-channel) or NMOS (n-channel) devices alone. Complementary circuitry (i.e., CMOS) allows simple one- and two-input gates to be used in latches and on-chip registers (see Figure 1-1). This provides a great advantage in power consumption as compared to NMOS without using significantly more area.

Circuits with highly complex, combinatorial gates employ n-channel MOS, with p-channel load devices. These pseudo-NMOS circuits dissipate more power than CMOS circuits but provide layout advantages for gates that have more than four inputs. Because the outputs of these circuits are transferred to static latches (flip-flops), the circuits do not consume static power.

A static device uses a flip-flop to store each bit and draws power continuously. A dynamic device uses a capacitance storage mechanism and must be refreshed (i.e., recharged) periodically to retain its data.

---

[1] *MAC-8 — MAC Tutor Reference Manual*, Bell Laboratories, August 1978.

[2] *PLAID Hardware Reference Manual*, Bell Laboratories, To be published.

[3] *PLAID Programmer's Manual*, Bell Laboratories, To be published.

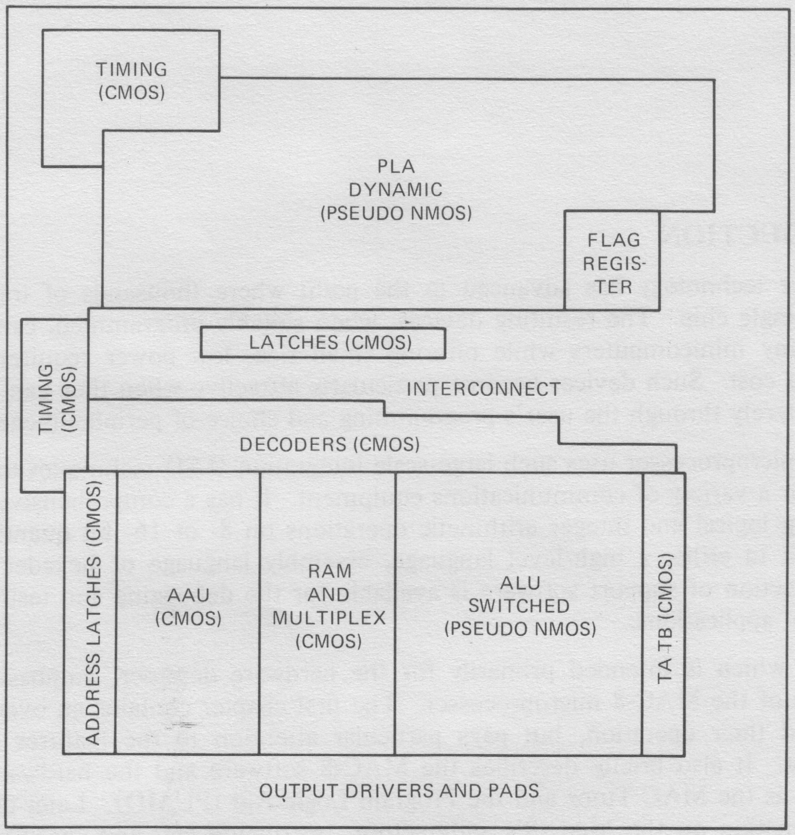

TIMING (CMOS)
PLA DYNAMIC (PSEUDO NMOS)
FLAG REGIS-TER
LATCHES (CMOS)
TIMING (CMOS)
INTERCONNECT
DECODERS (CMOS)
ADDRESS LATCHES (CMOS)
AAU (CMOS)
RAM AND MULTIPLEX (CMOS)
ALU SWITCHED (PSEUDO NMOS)
TA TB (CMOS)
OUTPUT DRIVERS AND PADS

Figure 1-1. Block Diagram of Chip

The logic arrays are accessed every cycle, hence dynamic circuits would be more suitable. In addition, CMOS would waste considerable area because these arrays have complex gates. Therefore, dynamic NMOS was selected for the logic arrays.

The appropriate mix of these techniques allows the overall chip power to be comparable to the power consumed by an all-CMOS chip. It also provides access speeds and gate density on the chip comparable to all-NMOS devices.

1.1.1 Microcomputer Operation

A microcomputer manipulates binary data according to a predefined set of instructions, a *program*, stored in its memory. It can be divided into three functional building blocks, as can any computer:

- *Memory* — stores information for the computer, such as the program, input data, and results.
- *Processor* — decodes and executes the instructions; that is, it performs calculations, makes decisions based on the results of calculations, and routes information between the other elements of the computer.
- *Input and Output (I/O) Devices* — transmit signals to the computer or receive signals representing the results.

One or more buses (a group of lines) interconnect these elements. Each line in a bus transfers one bit of information between the processor and memory or the processor and an I/O device or interface. Some microcomputers use a single bus for memory and I/O transfers.

The MAC-8 uses a 16-bit *address bus* for addressing both memory and I/O ports and an 8-bit bidirectional *data bus* for transferring data.

Memory consists of a group of consecutively numbered storage locations each holding 8 bits (or one byte) of information. Thus, each location has a unique memory address. A microcomputer often has two types of memory: read-only memory (ROM) and read/write or random-access memory (RAM).

As the name implies, the data in ROM can be read but not changed; that is, once programmed, the contents of ROM are permanently fixed and cannot be altered either by the microprocessor or the user. Some manufacturers do provide field-programmable ROMs (EPROMs), which can be erased only through external means and then reprogrammed. The microprocessor normally cannot alter such memory.

With RAM, data can be read from or written to memory at any time. This allows the user to program the computer for one application and change the program later for another application. The name *random access* is applied to such memory because the time required to access a memory location is independent of its address.

The MAC-8, for example, has 65,536 bytes of addressable memory space in which ROM, RAM, and I/O ports can be randomly located. At times, two consecutive memory locations or a pair of bytes are used to store a 16-bit operand or data word. The first memory location contains the low-order byte (bits 0 through 7) and the second location contains the high-order byte (bits 8 through 15 — hexadecimal 8 through F).

A processor generally is an integrated circuit (IC) that uses large-scale integration in metal-oxide semiconductor technology. The MAC-8 processor, which is fabricated using silicon-gate CMOS technology, is provided in a 40-pin dual in-line package (DIP). These pins provide the connection between the processor, its power source, and the external devices. Figure 1-2 shows the MAC-8 microprocessor chip in its ceramic package.

The chip measures 5.9 mm by 5.55 mm and contains roughly 7500 transistors. The DIP measures 2.02 inches long by 0.620 inch wide by 0.321 inch high (including the pin extensions).[4]

An I/O device, such as a printer or a terminal, usually cannot be connected directly to a processor. Instead, each device is connected to an I/O interface that converts information from the device into the signals required for the processor's data bus. An interface transforms the signals from an input device into the binary form required for the processor and converts the binary data from the processor into the proper form for the output device.

The I/O devices may be computer hardware — such as a printer, teletypewriter (TTY), and cathode-ray tube (CRT) — or components in the communications equipment that generate signals to the computer or receive signals from the computer. These signals enable the computer to monitor and control the equipment's operation.

1.1.2 Functional Elements

A *central processing unit* (CPU), which resides on the chip, is the heart of the microcomputer. The CPU consists of the circuitry necessary to access memory locations and to decode and execute instructions. This circuitry can be separated into several sections, each performing a different function (see Figure 1-3).

---

[4] Refer to **Package Dimensions** in Chapter 4.

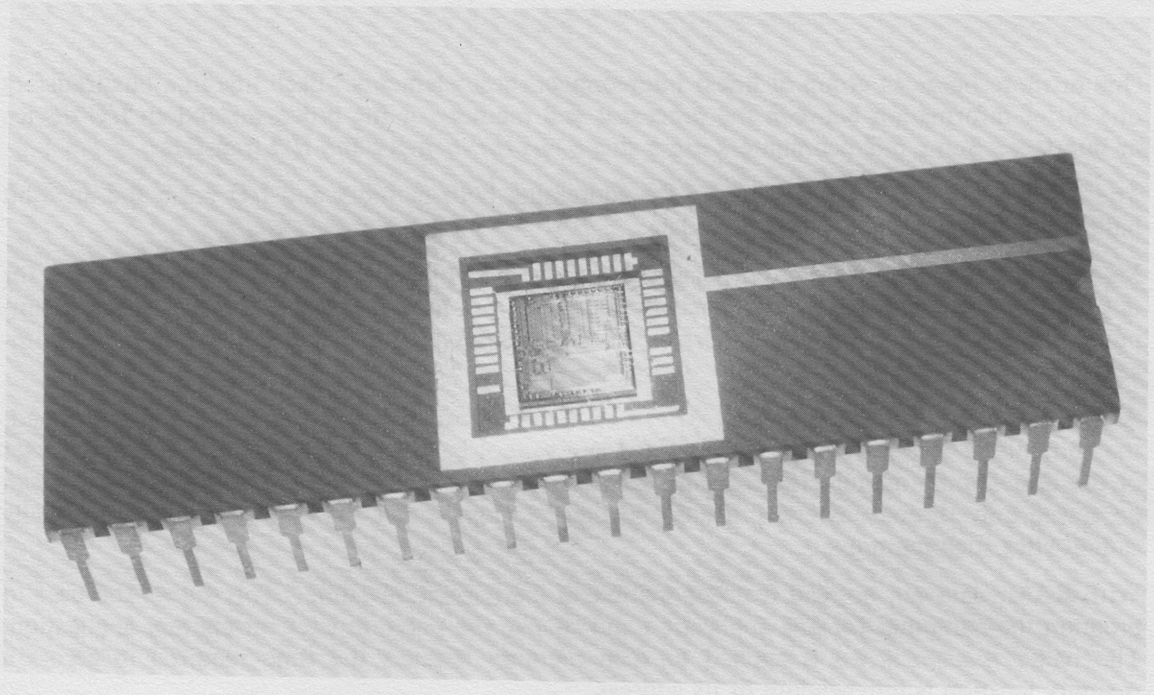


Figure 1-2. MAC-8 Microprocessor DIP

The CPU contains an *arithmetic logic unit* (ALU) that performs arithmetic operations, such as addition or subtraction, and logical operations, such as AND or OR.

In the MAC-8, two *programmed logic arrays* (PLA) form the control section of the CPU. A large PLA, the *instruction control logic array* (ICLA), directs the CPU circuitry through the states necessary to execute instructions. A smaller PLA, the *ALU control logic array* (ACLA), controls operation of the ALU and latches condition flags that the ALU generates. A separate *address arithmetic unit* (AAU) performs address calculations.

The CPU also contains registers for temporarily storing data and for manipulating data and instructions:

- *Program Counter* (PC) — contains the memory address of the instruction byte being executed or, upon completion of an instruction, contains the starting address of the next instruction to be executed.
- *Instruction Register* (IR) — contains the instruction being executed.
- *Condition Register* (CR) — contains bits to indicate conditions resulting from the operation just completed. For example, individual bits, when set, may indicate overflow from an operation, zeros or ones in the result, sign of the result, and carry (or borrow). A program instruction can test for the presence or absence of a condition to select the next instruction to be executed.

Some microprocessors (other than the MAC-8) also have the following registers in the CPU:

- *Memory address register* (MAR) — holds the address of the memory location to be accessed for an instruction or data.
- *Memory data register* (MDR) — stores the data that resides at the memory address identified by the MAR.

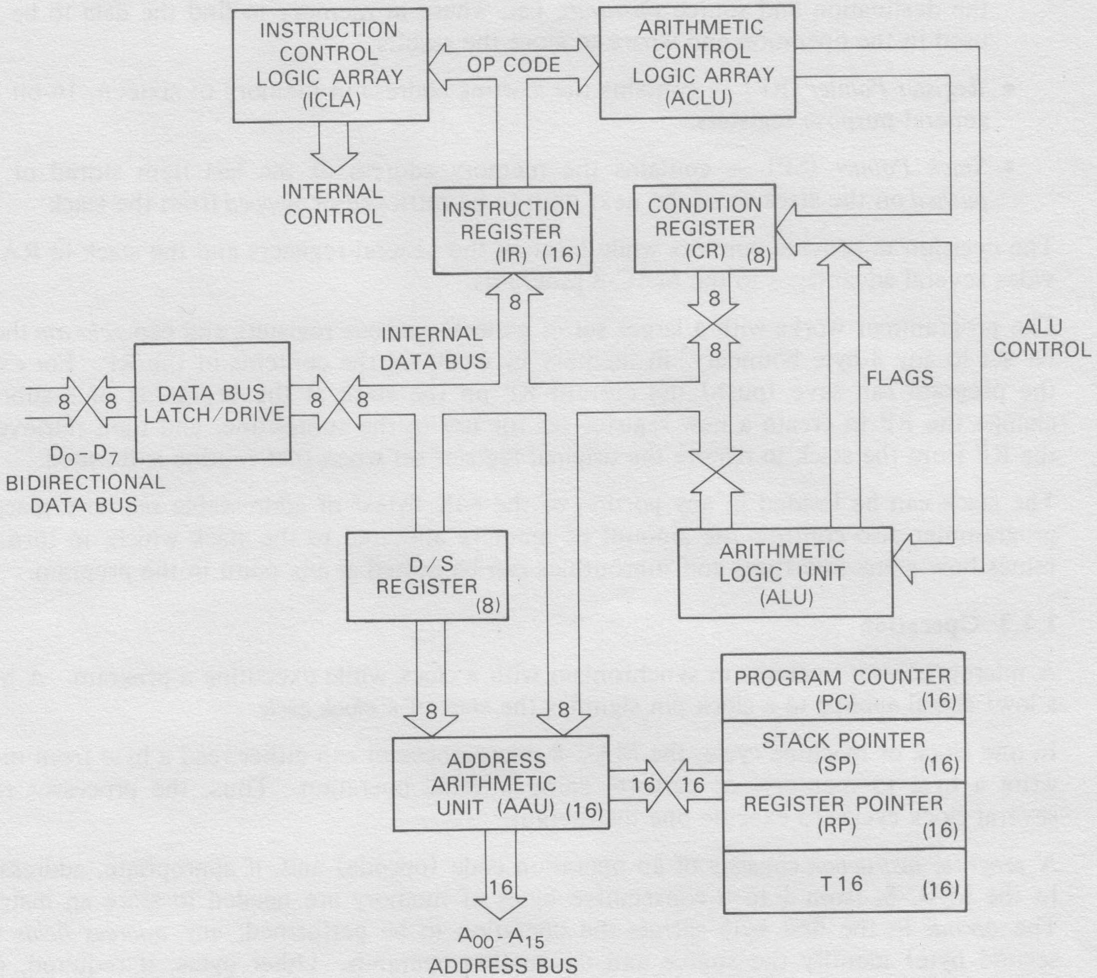

INSTRUCTION CONTROL LOGIC ARRAY (ICLA)
OP CODE
ARITHMETIC CONTROL LOGIC ARRAY (ACLU)
INTERNAL CONTROL
INSTRUCTION REGISTER (IR) (16)
CONDITION REGISTER (CR) (8)
8
8
8
ALU CONTROL
INTERNAL DATA BUS
FLAGS
8
8
DATA BUS LATCH/DRIVE
8
8
D0 - D7
BIDIRECTIONAL DATA BUS
ARITHMETIC LOGIC UNIT (ALU)
D/S REGISTER (8)
PROGRAM COUNTER (PC) (16)
STACK POINTER (SP) (16)
8
8
ADDRESS ARITHMETIC UNIT (AAU) (16)
16
16
REGISTER POINTER (RP) (16)
T16 (16)
16
A00 - A15
ADDRESS BUS

Figure 1-3. Functional Diagram of MAC-8 CPU

- *Working or general-purpose registers* — one or more registers the program uses for indexing (or addressing) or as accumulators in arithmetic or logical operations.
- *Stack* — an array of registers that allows addresses or data in the array to be accessed (stored or retrieved) on a last-in, first-out basis. The stack enables the processor to leave an instruction sequence, execute a separate set of instructions, and then return to the point at which the instruction stream was broken. This capability permits the processor to respond to *interrupts* (requests for service from external devices) or to call *subroutines* (a series of instructions that may be executed frequently in a program) and then resume its original activity.

Because the size of the chip limits the number of on-chip registers, the MAC-8 designers took a different approach. In the MAC-8 microprocessor, the AAU uses the address bus and data bus to perform the same tasks as the MAR and MDR. The designers also elected to locate the MAC-8's general-purpose registers and stack in random access memory (RAM). They then placed the following registers on the chip:

- *T16* — used as a scratchpad register during address calculations.

- *D/S Register* — stores two 4-bit qualifiers, one for the destination (D) and the other for the source (S). These qualifiers are used for determining the address of the destination and source *operands*, i.e., where in memory to find the data to be used in the operation and where to store the results.
- *Register Pointer* (RP) — contains the starting address in memory of sixteen, 16-bit general-purpose registers.
- *Stack Pointer* (SP) — contains the memory address of the last item stored or *pushed* on the stack or of the next item to be retrieved or *popped* from the stack.

The decision to provide pointers while locating the general registers and the stack in RAM provides several advantages to the MAC-8 programs.

The programmer works with a larger set of general-purpose registers and can *relocate* the register set to any 8-byte boundary[5] in memory by changing the contents of the RP. For example, the program can save (push) the current RP on the stack at the beginning of a subroutine, change the RP to create a new register set for use in the subroutine, and then retrieve (pop) the RP from the stack to restore the original register set when that routine is finished.

The stack can be located in any portion of the 64K bytes[6] of addressable memory space. The programmer also controls the amount of memory allocated to the stack which, in turn, determines how many interrupts and subroutines can be nested at any point in the program.

1.1.3 Operation

A microprocessor operates in synchronism with a clock while executing a program. A high (or a low) signal applied to a clock pin signifies the start of a *clock cycle*.

In one clock or machine cycle, the MAC-8 microprocessor can either read a byte from memory, write a byte to memory, or perform some internal operation. Thus, the processor requires several clock cycles to execute one instruction.

A *program instruction* consists of an operation code (opcode) and, if appropriate, address fields. In the MAC-8, from 1 to 6 consecutive bytes of memory are needed to store an instruction. The *opcode* in the first byte selects the operation to be performed; any *address fields* (in the second byte) identify the source and destination operands. Other bytes, if required, contain data for the operation or addressing.

Several clock cycles form an *instruction cycle* during which the processor performs memory reads, memory writes, and internal operations to fetch and execute an instruction. While executing one instruction, the CPU goes through the following states:

- *Opcode fetch* — The CPU fetches the first byte of the instruction, places it in the IR, and decodes the byte.
- *Operand fetch* — If appropriate, the CPU fetches the second byte of the instruction, places it in the D/S Register, and decodes the byte.
- *Execute* — The CPU executes the selected function. During this phase, memory reads and writes may be performed to fetch data used in the operation and to store the results.

At the start of an instruction cycle, the PC points to the memory location that contains the first byte of the instruction to be executed. After each fetch of an instruction byte, the PC is automatically incremented; hence it always points to the next byte to be fetched for the current instruction or to the first byte of the next instruction to be executed.

---

[5] The starting address must be divisible by 8.

[6] One K is 1024, hence 64K bytes is equivalent to 65,536 bytes.

In the MAC-8 instruction set, the first instruction byte contains the opcode and, if appropriate, the address mode indicator. The address mode tells the AAU how to compute the memory addresses for the operands (identified in the second instruction byte). Some instructions have two operands — one designated the source and the other the destination — and are referred to as *dyadic* instructions. Both operands provide data for the operation but, because the results are stored at the destination location, only the contents of that memory address are changed. Some instructions have only one operand that is both the source and the destination. These are the *monadic* instructions.

The opcode tells the CPU whether the operand fetch involves one or two operands. After fetching the operand byte and computing the source and destination addresses, the CPU enters the execute phase. During this phase, data is read from memory, the operation is performed, and the results are stored.

The instructions normally are executed in the sequence in which they are stored in memory. Frequently, the current operation is a control transfer instruction that may transfer execution to another point in the program. Such instructions (CALL, RETURN, and BRANCH) can take two forms:

- *Unconditional* — transfer to another location in the program always occurs
- *Conditional* — transfer occurs only if conditions specified in the instruction are met.

A conditional BRANCH instruction, for example, checks the setting of one or more bits in the CR. If the condition is satisfied, the program branches to the location specified in the instruction. That is, the CPU sets the PC to the address of an instruction at some other point in the program and that instruction is executed next. If the condition is not met, program execution continues with the next instruction in memory.

The situation is slightly different when executing a CALL or a RETURN. Suppose a particular task is to be performed several times during a program. The required instructions could be repeated each time the task must be performed, but this uses up memory space. A more efficient method is to group the instructions for this task into a subordinate program called a *subroutine* and to call the subroutine at appropriate points in the main program. When a CALL is executed, the CPU automatically saves the PC on the stack and sets the PC to the address of the first instruction in the subroutine. The RETURN automatically retrieves the saved address from the stack and places it in the PC. Processing then continues with the instruction that would have been executed next if the call had not occurred.

External devices frequently affect the instruction sequence by requesting interrupts. An *interrupt* is a break in the normal instruction sequence that enables the processor to provide service to an external device. An external device requests an interrupt when it is ready to accept output from the processor or transmit input data.

In the MAC-8, a bit in the CR must be set to *enable* interrupts; if the bit is off, interrupts are not accepted. When this bit is set, the processor checks for an interrupt request at the end of each instruction cycle. If no interrupt-request signal is present, processing continues with the next instruction. However, when a signal is present, the processor saves the PC and CR on the stack, disables further interrupts, and starts the interrupt instruction sequence. A RETURN FROM INTERRUPT instruction, at the end of this instruction sequence, restores the PC and CR and enables interrupts; processing continues with the original instruction sequence, or another interrupt may be accepted.

Usually, all data must pass through the CPU when being transferred from input to memory or from memory to output. This process can be time-consuming for large arrays of data because several program instructions are required to transfer each data byte or data word. Some microprocessors, such as the MAC-8, provide the capability for external devices to make *direct memory access* (DMA) transfers; that is, the external device can access memory directly without accessing the CPU. Generally, the device signals the processor to gain access for a DMA operation. The CPU checks for this signal at the end of each machine cycle. If it is found, the processor acknowledges the DMA request, passes control of the buses to the external device, and temporarily suspends its own operation. Normal processing resumes when the external device signals the end of the DMA operation.

For additional information about the architecture of the MAC-8 microprocessor, refer to Chapter 2. The remainder of this chapter describes the MAC-8 development tools.

1.2 Software Development System

An integrated set of software development tools is available for the MAC-8 microprocessor. Figure 1-4 identifies the tools that comprise the software development system. The C compiler, structured assembler, flexible link editor, and source-oriented simulator reside on the UNIX* system and are described in detail in the *MAC-8 Programmer's Manual*. A source-oriented debugger resides on a hardware prototyping system called PLAID (Program Logic Aid) and is described in detail in the *PLAID Programmer's Manual*.

The use of C-syntax input language, UNIX file-oriented intertool communication, and names analogous to those of the corresponding UNIX tools (e.g., m8cc and m8as) provides a consistent user interface to MAC-8's software development tools.

C Language

The high-level language C gives the microprocessor system designer control over the resources of the machine while retaining the expressive power of a high-level language.

Features that provide direct control over the processor's resources include:

- Specification of primitive types of variables such as
  - ☐ *character variables* to manipulate 8-bit portions of memory
  - ☐ *pointer variables* to manipulate addresses
  - ☐ *register variables* to specify which variables are to reside in registers.
- Efficient address manipulation using machine-level operations such as indirection, increment, and decrement.
- Efficient bit manipulation using the usual arithmetic operators and Boolean, shift, and rotate operators.

The C language also has most of the advantages of a high-level language:

- Allocation of automatic, static, and register storage and the ability to allocate storage for data aggregates such as arrays and structures
- Evaluation of expressions
- A function-calling mechanism that includes passing arguments, saving and restoring registers, and returning values
- Control constructs such as *if-else*, *switch*, *do*, and *while*.

---

* Trademark of Bell Laboratories.

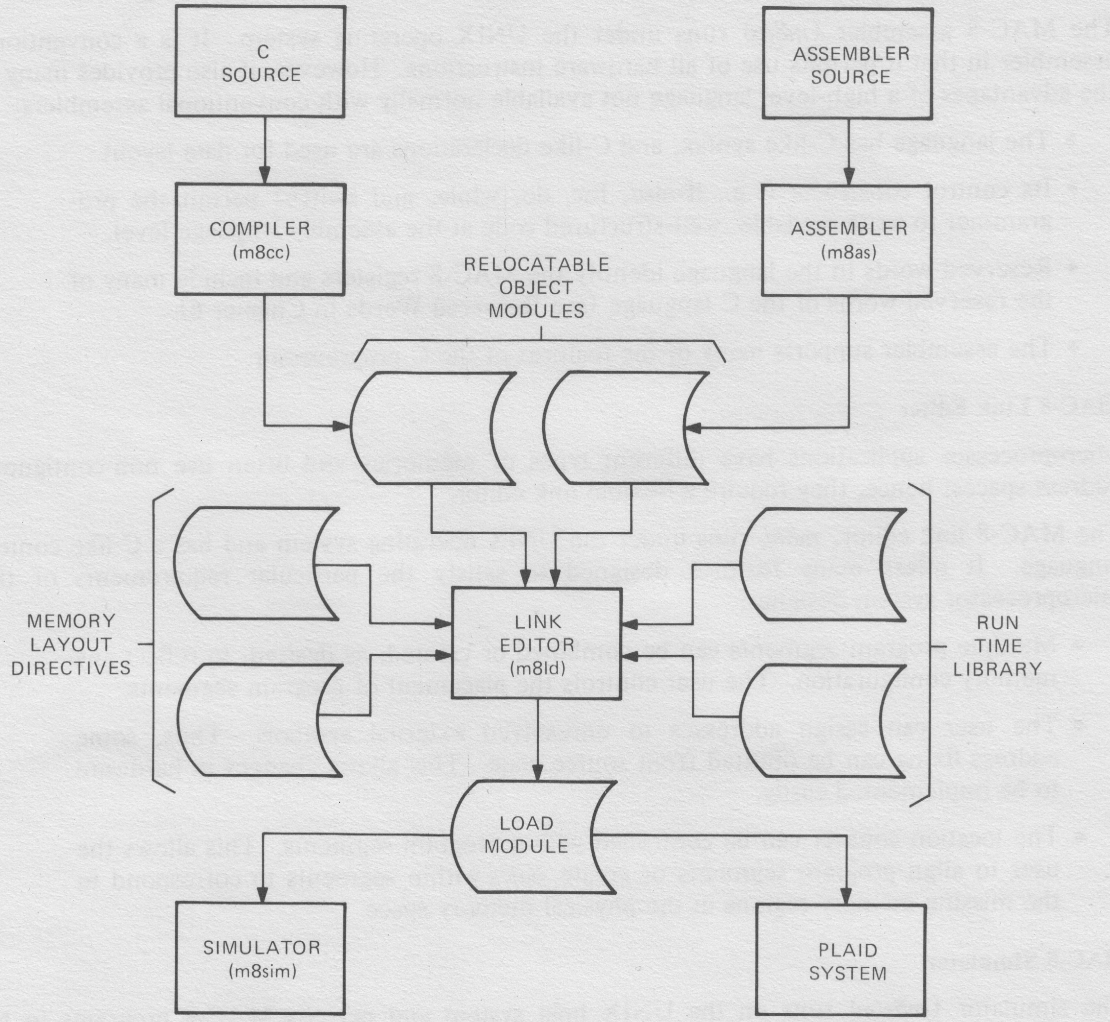

C SOURCE
ASSEMBLER SOURCE
COMPILER (m8cc)
ASSEMBLER (m8as)
RELOCATABLE OBJECT MODULES
MEMORY LAYOUT DIRECTIVES
LINK EDITOR (m8ld)
RUN TIME LIBRARY
LOAD MODULE
SIMULATOR (m8sim)
PLAID SYSTEM

Figure 1-4. MAC-8 Software Support

MAC-8 C Compiler

The C cross-compiler (*m8cc*) runs under the UNIX operating system and allows the user to write a well-structured, easily read, and easily maintained program. Usually several algorithms can be written in a high-level language in the time it takes to implement one algorithm in assembly language.

To use the compiler, the user needs a full-ASCII terminal for interacting with the UNIX system and a file that contains the C source-language program. From this file, the compiler creates an absolute binary file that reflects the contents of the C source file and any library subroutines it calls. This binary file can be used with the MAC-8 simulator (*m8sim*) to simulate the program, or it can be loaded and debugged (with the user's hardware) with the debugging tool, PLAID.

MAC-8 Assembler

The MAC-8 assembler (*m8as*) runs under the UNIX operating system. It is a conventional assembler in that it permits use of all hardware instructions. However, it also provides many of the advantages of a high-level language not available normally with conventional assemblers:

- The language has C-like syntax, and C-like declarations are used for data layout.
- Its control constructs (e.g., if-else, for, do, while, and switch) permit the programmer to write readable, well-structured code at the assembly language level.
- Reserved words in the language identify the MAC-8 registers and include many of the reserved words of the C language (see **Reserved Words** in Chapter 6).
- The assembler supports many of the features of the C preprocessor.

MAC-8 Link Editor

Microprocessor applications have different types of memories and often use non-contiguous address spaces; hence, they require a flexible link editor.

The MAC-8 link editor, *m8ld*, runs under the UNIX operating system and has a C-like control language. It offers many features designed to satisfy the particular requirements of the microprocessor system designer:

- Multiple program segments can be combined or created, as desired, to reflect any memory configuration. The user controls the placement of program segments.
- The user can assign addresses to unresolved external symbols. Thus, some address fixing can be omitted from source code. This allows changes in hardware to be implemented easily.
- The location counter can be controlled within program segments. This allows the user to align program segments or create *holes* within segments to correspond to the missing memory regions in the physical memory space.

MAC-8 Simulator

The simulator (*m8sim*) runs on the UNIX host system and permits MAC-8 programs to be debugged without using MAC-8 hardware. It provides software simulation of the MAC-8 processor using symbolic debugging at the source level, i.e., with either C compiler or assembler source code.

The simulator offers the following capabilities to the user:

- Program control features
    - ☐ Load a program into memory.
    - ☐ Set memory boundaries.
    - ☐ Start, stop, and step the program.
    - ☐ Derive timing information on the program.
- Breakpoint features
    - ☐ Define breakpoints on program statements and/or memory locations.
    - ☐ Specify actions when the break occurs. Such specifications can include displaying memory, registers, and the stack. Conditional and looping

constructs can control the sequence of actions.

- Command language features
  - ☐ Redirect input and output interactively.
  - ☐ Define new commands in terms of other commands.
- Symbolic debugging features
  - ☐ Reference C identifiers, both local and global.
  - ☐ Reference C source statements by line number; numbers are on a per-function basis.

Utilities

The MAC-8 system has utilities (like the UNIX system):

- *m8size* — determines the size of an object file
- *m8nm* — determines the contents of the symbol table
- *m8list* — disassembles code and lists function line numbers
- *m8hx* — formats an object file to permit *downline* loading into the MAC-8 based application.

1.3 Hardware Development Systems

Two hardware support systems are available to assist MAC-8 users. The *MAC Tutor* is a self-teaching machine that introduces the prospective user to the scope and capabilities of the MAC-8 microprocessor. The *PLAID* development system enables a user to test, debug, and correct MAC-8 programs using the application's hardware.

The remainder of this chapter summarizes the features of the MAC Tutor and PLAID. Both systems are supported by their own sets of documentation.

MAC Tutor

The MAC Tutor is a low cost, self-contained, microprocessor based system developed to familiarize prospective users with the essentials of microprocessor operation and, in particular, the MAC-8 microprocessor. It contains an on-board keypad with an 8-digit light-emitting diode (LED) display that permits the user to:

- enter and examine instructions and data in memory
- execute an entire stored program or, if desired, single-step program instructions
- examine the results.

Initially, the user employs only the basic MAC Tutor hardware to become familiar with the design criteria and tradeoffs that are inherent in a useful computer. After gaining experience, the user can easily add peripheral equipment (such as Teletype or cassette recorder) to expand the Tutor's capability for hardware and software development.

The MAC Tutor is equipped with the following features identified in Figure 1-5:

- MAC-8 microprocessor, with access to 64K bytes of addressable memory space.
- 2K-byte ROM executive program that controls all of the hardware features. This program contains the routines needed to drive the display, read the keyboard, communicate with a terminal, etc. It enables the user to store, execute, and debug programs.

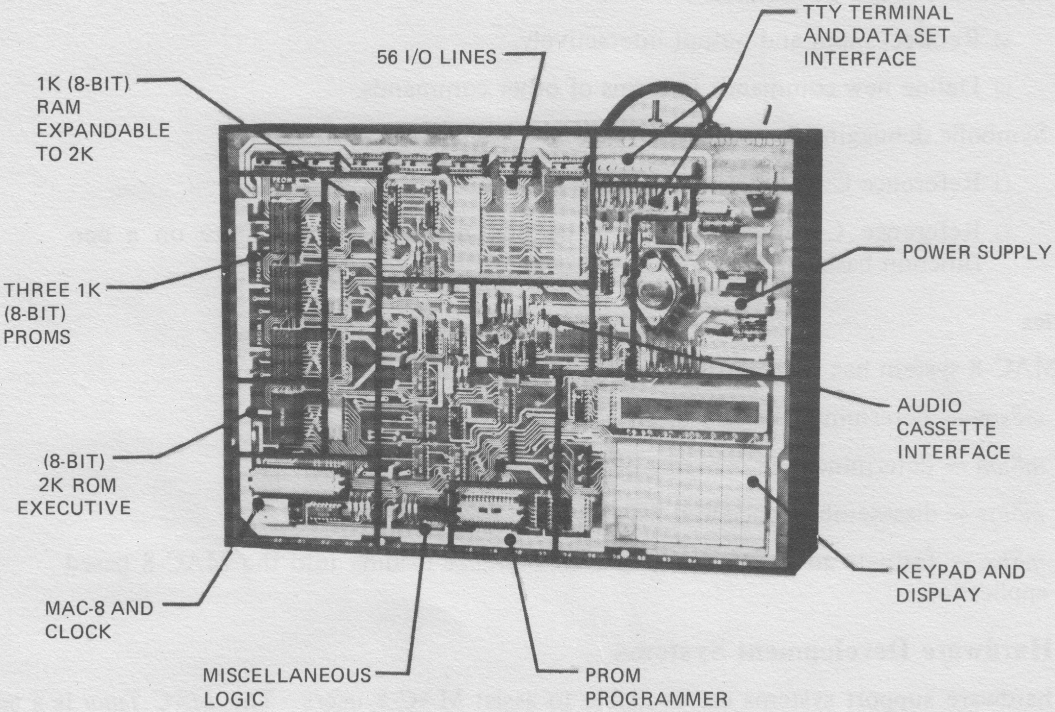

TTY TERMINAL AND DATA SET INTERFACE
56 I/O LINES
1K (8-BIT) RAM EXPANDABLE TO 2K
POWER SUPPLY
THREE 1K (8-BIT) PROMS
AUDIO CASSETTE INTERFACE
(8-BIT) 2K ROM EXECUTIVE
KEYPAD AND DISPLAY
MAC-8 AND CLOCK
MISCELLANEOUS LOGIC
PROM PROGRAMMER

Figure 1-5. The MAC Tutor

- Sockets for three 1K-byte 2708 EPROMs that provide permanent storage for the user's programs. These electrically programmable read-only memories can be erased by exposure to ultraviolet light.
- 1K bytes of RAM, with sockets to expand to 2K bytes. This section of memory is used for the MAC-8 registers, stack memory, and user's program. Because this memory is volatile (i.e., the contents are destroyed when power is removed from the MAC Tutor), its contents must be recorded into PROM or onto cassette tape if the data is to be retained.
- 32 input/output lines, with a socket to add another 24 lines. Address and data buses for these I/O lines are available on 16-pin connectors at the periphery of the MAC Tutor and permit adding peripherals or extra memory. The Tutor also has 32 I/O lines, which it uses for internal I/O.
- TTY terminal and data set interface. The Tutor can communicate with a TTY terminal through an EIA RS-232C interface, thus extending all functions of the on-board keypad and display to a terminal. The interface to the terminal is under direct control of the MAC-8 through a serial I/O line. The executive program performs all timing and parallel-to-serial conversion, thus allowing the MAC Tutor to adapt to the baud rate of the terminal (i.e., 0 through 2400 baud). The Tutor can accept data from a remote computer through telephone lines when the modem connection is made. An on-board computer/TTY data switch allows the

user to connect the terminal to the modem with the MAC Tutor in the listening mode or to the Tutor with the modem switched out.

- On-board power supply. The MAC Tutor plugs into a conventional 110V, 60-Hz outlet to generate the voltages required for the on-board PROM programmer, the logic, and the 2708 EPROMs.
- Audio cassette interface. This interface provides alternate file storage using cassette tape. Any standard, audio-grade cassette recorder can be used. When connected to the recorder microphone input and earphone output, the MAC Tutor can store and retrieve data in files on the cassette tape at a rate of approximately 16 bytes per second.
- Keypad and display. The 28-button keypad consists of a 4-by-7 array of switches used for entering program instructions and data. Each key on the keypad represents a command. The display consists of eight 7-segment LED arrays, each capable of displaying one hexadecimal digit. These LED arrays display memory addresses and the contents of memory locations.
- PROM programming socket. This 24-pin socket enables the user to store MAC-8 programs on a 2708 1K-byte EPROM.

PLAID

PLAID is a hardware/software system that serves as the primary development tool for MAC-8 based applications. With PLAID, the user's program runs in the application's environment where the user can modify the inputs and monitor the program's behavior. PLAID debugs programs written in either assembly language or the high-level C language.

The hardware includes two MAC-8 systems connected by a hardware monitor system. This configuration (see Figure 1-6) permits one MAC-8 system, the *master*, to closely monitor and control the other MAC-8, the *slave*. Each MAC-8 has separate I/O, which allows connection to peripheral devices from the master and to application hardware from the slave. The monitor hardware includes debugging aids and the MAC Cable, which allows in-circuit control of any MAC-8 system. This cable has a 40-pin connector that clamps directly over the user's MAC-8 to monitor the user system or into the MAC-8's socket to give in-circuit control of the user's application.

The software system includes:

- The *Satellite Processing System* (SPS) that communicates with a parent UNIX system, which performs operating system functions for the master and monitor hardware.
- A source-oriented symbolic debugger *m8db* that permits a user to reference either compiler or assembler source program symbols and then symbolically control and monitor the load module. The debugger also allows conditional execution of prestored procedures or commands. User-program and debug symbols can specify C expressions, which makes it possible to compose complex debugging experiments in an interactive fashion.

Figure 1-6 illustrates the PLAID hardware system. The master consists of a MAC-8 (with a 32K-byte PROM and 16K-byte RAM) and interfaces to a user terminal, a dial-up link to the UNIX system, and other I/O ports. The master uses DMA to interact with the slave system and monitors the slave with the assistance of the monitor system.

The slave consists of a MAC-8 with a 65K-byte RAM, 36 I/O ports, and the MAC Cable.

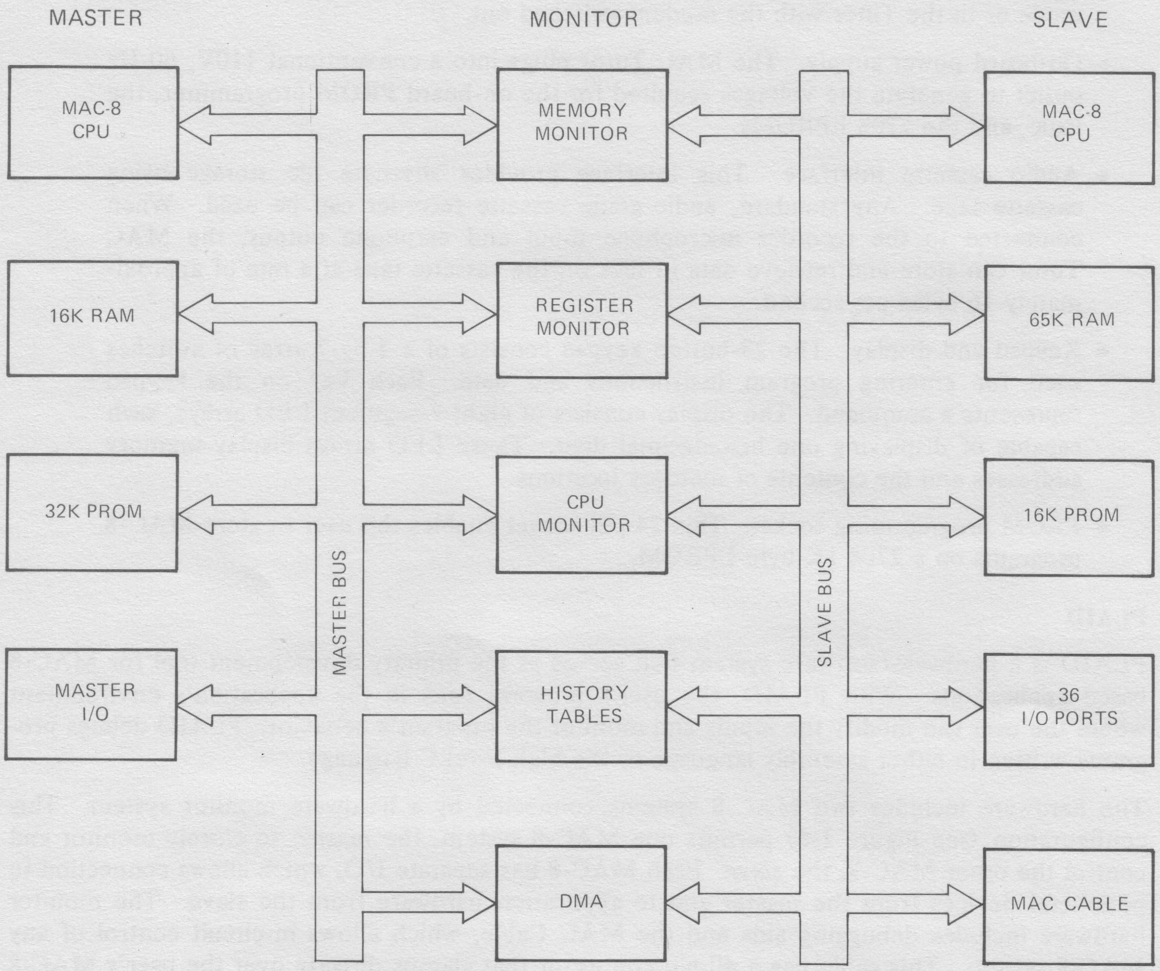

MASTER
MONITOR
SLAVE
MAC-8 CPU
16K RAM
32K PROM
MASTER I/O
MASTER BUS
MEMORY MONITOR
REGISTER MONITOR
CPU MONITOR
HISTORY TABLES
DMA
SLAVE BUS
MAC-8 CPU
65K RAM
16K PROM
36 I/O PORTS
MAC CABLE

Figure 1-6. PLAID Hardware System

The PLAID monitor system can watch the actions of the slave or user system. This allows MAC-8 applications to be debugged in a real-time environment. Its features include:

- *Memory monitor* — contains a 65K-byte by 4-bit dynamic RAM that enables breakpointing based on a variety of conditions such as:
  - ☐ memory read, write, or reference
  - ☐ opcode fetch
  - ☐ arbitrary 8-bit pattern in memory.
- *CPU monitor* — contains shadow registers that hold the current values of the slave/user MAC-8 on-chip registers (CR, SP, RP, PC) and provide breakpoint capability on updates.
- *Register monitor* — permits breakpointing at a read or write of the MAC-8 off-chip registers.

- *Jump trace* — contains a history table of the last 32 Program Counter discontinuities (i.e., pairs of from/to PC values).
- *Instruction trace* — consists of sixty-four 4-byte entries containing the last 64 clock cycles executed by the slave.
- *Memory access circuitry* — permits the user to selectively specify the read/write protect and the number of wait states to be generated for memory accesses on a block basis (256 bytes per block).
- *Clock frequency* — enables the user to specify a clock frequency for the slave from a list of predefined frequencies.

Chapter 2

ARCHITECTURE

2. ARCHITECTURE

The MAC-8 microprocessor is provided in a 40-pin DIP. A 16-bit address bus allows direct addressing of 65,536 bytes of memory.

The instruction repertoire includes logical and integer arithmetic operations on 8- or 16-bit quantities, supplemented by 16-bit operations to facilitate address arithmetic. A rich set of addressing modes enhances memory addressing capability. The instructions and addressing modes are described in Chapter 3.

The MAC-8 accesses a set of working registers, located in external random access memory (RAM). These registers, together with the addressing modes, form a highly symmetrical set of operand combinations for the instructions.

A pushdown stack, also located in RAM, serves as the call and return mechanism for subroutines. The processor can respond to unusual events (such as interrupt requests) and uses the pushdown stack to save and restore internal machine status.

2.1 Pin Descriptions

Figure 2-1 identifies the functions of the 40 pins of the MAC-8 DIP. A bar over the word(s) that identify a pin's function — for example, $\overline{\text{READ}}$ — indicates a low transistor-transistor logic (TTL) level signal on the pin activates that function.

The 40 pins of the MAC-8 DIP have the following functions:

A_{00}–A_{15} (Input/Output 3-state)

The 16-bit *address bus* selects the appropriate memory location or I/O port. For a bus loading of 20 pF, the address is valid 60 ns after the start of each cycle. The bus assumes a high-impedance state for a DMA transfer (i.e., when $\overline{\text{DMA ACK}}$ is active) or for a HALT state of the MAC-8.

CLK 0, CLK 1 (Input)

CLK OUT (Output)

The MAC-8 has an internal clock or can be driven by an external clock. When connected to an external resonator, CLK 0 and CLK 1 enable the internal clock generator; CLK OUT provides a clock output signal to external devices. The internal clock rate cannot exceed 2 MHz. Alternatively, an external clock input signal can be applied to the CLK 1 pin; in this case, the CLK 0 pin is not used. The CLK OUT signal is also provided in this mode of operation.

A *clock cycle* starts when the CLK OUT signal makes a high to low transition.

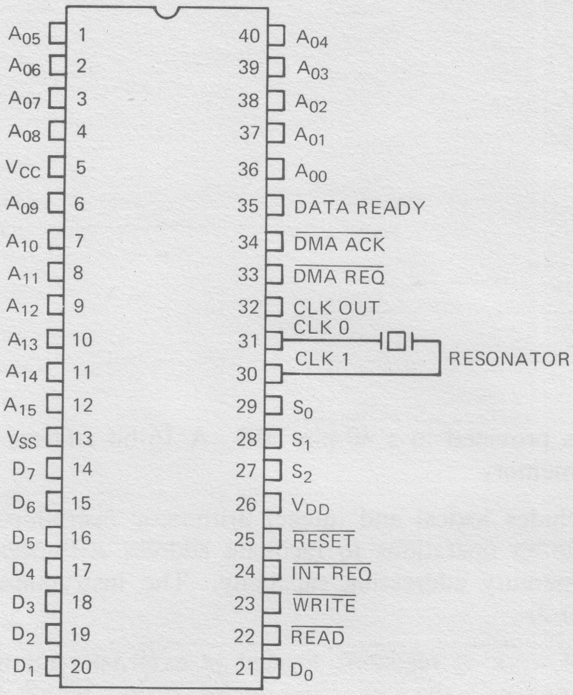

A05 1
A06 2
A07 3
A08 4
VCC 5
A09 6
A10 7
A11 8
A12 9
A13 10
A14 11
A15 12
VSS 13
D7 14
D6 15
D5 16
D4 17
D3 18
D2 19
D1 20
40 A04
39 A03
38 A02
37 A01
36 A00
35 DATA READY
34 DMA ACK
33 DMA REQ
32 CLK OUT
31 CLK 0
30 CLK 1
RESONATOR
29 S0
28 S1
27 S2
26 VDD
25 RESET
24 INT REQ
23 WRITE
22 READ
21 D0

Figure 2-1. MAC-8 Pin Assignments

D_0–D_7 (Input/Output 3-state)

The 8-bit *data bus* is used for memory or I/O data transfer. The $\overline{\text{READ}}$, $\overline{\text{WRITE}}$, and DATA READY pins control such transfers. Data on the bus is strobed into an internal latch at the end of a READ cycle. In a WRITE cycle, the bus data is valid at mid-cycle. The DATA READY signal must be active (high logic level) when checked at mid-cycle[1] for a data transfer to occur on a READ or a WRITE operation. The bus assumes a high-impedance state for a DMA transfer (i.e., when $\overline{\text{DMA ACK}}$ is active), for a HALT condition, and during some non-access cycles.

DATA READY (Input)

The DATA READY input signal enables the CPU to synchronize slow memory or I/O devices. An active (high logic level) signal indicates the answering device is ready for data transfer on a READ or WRITE operation. The CPU polls the DATA READY pin at mid cycle. If the signal is true (high), the $\overline{\text{READ}}$ (or $\overline{\text{WRITE}}$) pulse terminates at the end of the cycle. If the signal is not true at mid-cycle, the $\overline{\text{READ}}$ (or $\overline{\text{WRITE}}$) signal remains active (low) until the next mid-cycle check when the process is repeated.

---

[1] Refer to **Access Timing** in Chapter 5.

$\overline{\text{DMA REQ}}$ (Input)
$\overline{\text{DMA ACK}}$ (Output)

An active (low logic level) $\overline{\text{DMA REQ}}$ signal indicates to the CPU that an external device wants access to the bus for a DMA operation. The CPU polls $\overline{\text{DMA REQ}}$ at the end of each clock cycle. If the signal is active and the chip is not in the *wait* state (i.e., DATA READY is high), the CPU acknowledges the request at the beginning of the next internal cycle, i.e., the $\overline{\text{DMA ACK}}$ signal becomes active (low logic level). The acknowledge passes master bus control to the requesting device; the buses and control leads are in the high-impedance mode.

$\overline{\text{INT REQ}}$ (Input)

An active (low logic level) $\overline{\text{INT REQ}}$ signal indicates to the CPU that an external device is requesting an interrupt. The CPU polls $\overline{\text{INT REQ}}$ during the last clock cycle of each instruction cycle. The request is accepted if the interrupt enable condition is set; otherwise, the request is ignored. When acknowledging the interrupt request, the CPU clears the interrupt enable condition, pushes PC and CR onto the stack, and addresses memory location[2] X(FFFF). It then reads the data byte X(yy) which the interrupting device supplies on the data bus. The CPU uses this byte to form a pointer, the interrupt vector X(00yy), to the low (0 through 256) bytes of memory. $\overline{\text{INT REQ}}$ must remain active until after the request is acknowledged by the read from address X(FFFF).

$\overline{\text{READ}}$ (Output)

An active (low logic level) $\overline{\text{READ}}$ clock signal indicates a READ memory access is in progress. The $\overline{\text{READ}}$ pulse lasts until the end of the machine cycle and remains active until a non-READ cycle occurs. The contents of the data bus are strobed into an internal latch at the end of the READ cycle.

$\overline{\text{RESET}}$ (Input)

An active (low logic level) $\overline{\text{RESET}}$ signal causes the MAC-8 to be reset. The CPU pushes the PC and CR onto the stack, clears the interrupt enable condition, clears the PC, and acknowledges the RESET request by a read from address X(FFFF). After RESET, instruction execution starts at memory location X(0000).

S_0,S_1,S_2 (Output)

These *Status* pins signal various internal states of the processor (refer to **CPU Status**).

V_{CC} (Input)
V_{DD} (Input)
V_{SS} (Input)

Power to the chip is supplied on pin V_{DD} and can vary from +8.6V to +12.6V. The lower voltages produce a relatively slow version of the chip (i.e., slow memory accessing). The 2-MHz version of the chip, with automatic *power-on reset*, requires +12V ± 5% on this pin. A separate

---

[2] This manual uses the notation X(n) to denote the hexadecimal number n.

+5V power supply operates the on-chip TTL interface circuitry. This power is supplied on pin V_{CC}. Pin V_{SS} is the ground (0V). To obtain power-on reset, power must be applied first to pin V_{DD} and then to pin V_{CC}. At 2 MHz, the expected total power dissipation is below 100 mW.

$\overline{\text{WRITE}}$ (Output)

An active (low logic level) $\overline{\text{WRITE}}$ clock signal indicates a WRITE memory access. The active $\overline{\text{WRITE}}$ pulse is delayed 150 ns (nominal) from the beginning of the WRITE clock period and terminates 5 ns before the end of the WRITE cycle. Data on the data bus is valid at mid-cycle.

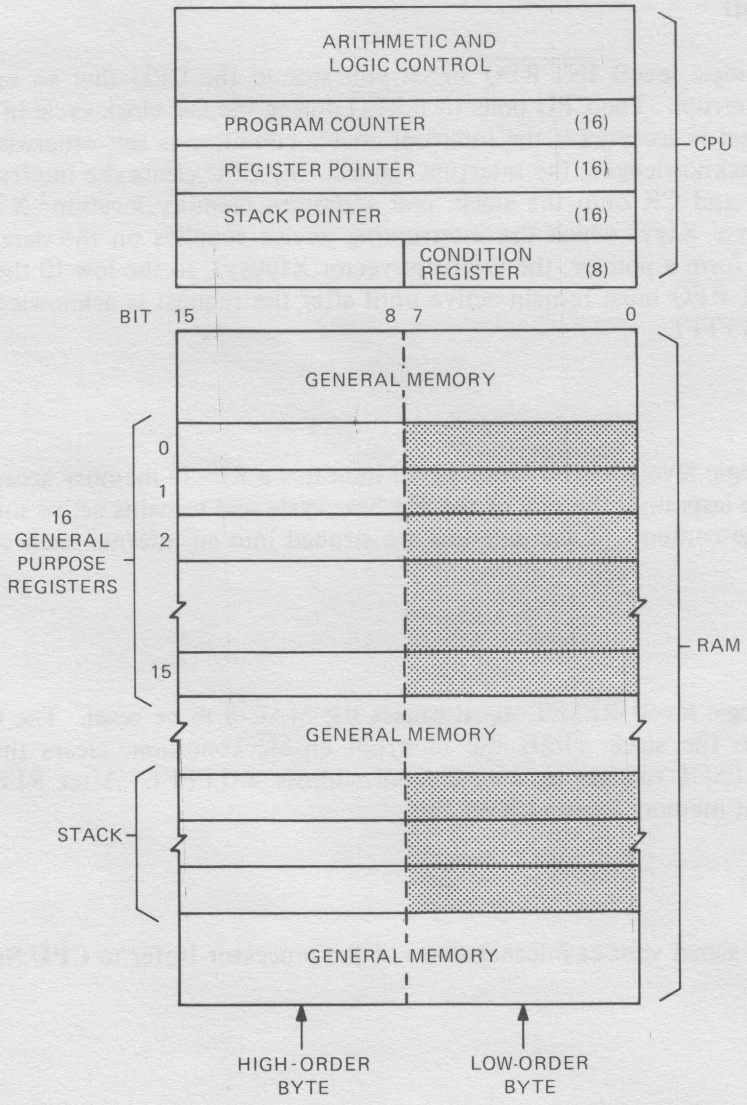

ARITHMETIC AND LOGIC CONTROL
PROGRAM COUNTER (16)
REGISTER POINTER (16)
STACK POINTER (16)
CONDITION REGISTER (8)
CPU
BIT 15 8 7 0
GENERAL MEMORY
0
1
2
15
16 GENERAL PURPOSE REGISTERS
RAM
GENERAL MEMORY
STACK
GENERAL MEMORY
HIGH-ORDER BYTE
LOW-ORDER BYTE

Figure 2-2. CPU and RAM Registers

2.2 Registers

The MAC-8 registers are located either on the chip or in external RAM (see Figure 2-2). The on-chip registers are used for program control or memory pointers while those in memory are used for data storage or indexing.

General Registers

The MAC-8 provides 16 *general-purpose registers* that the program can locate in any 32-byte section of available memory (RAM); the only restriction is the starting address must be at an 8-byte boundary — i.e., it must be divisible by 8. The first two bytes are register 0, the next two are register 1, and so on (see Figure 2-2). Each of these registers can be accessed either as a 16-bit base or *B Register* to hold memory addresses or 16-bit operands, or as an 8-bit accumulator or *A Register* for arithmetic and logical operations (see Figure 2-3). When the executing program accesses any of these registers as an A Register, only the low-order byte participates in the operation; the high-order byte remains unchanged. Some operations such as addition can be performed on either A or B Registers; others, such as negation, can be performed only on A Registers.

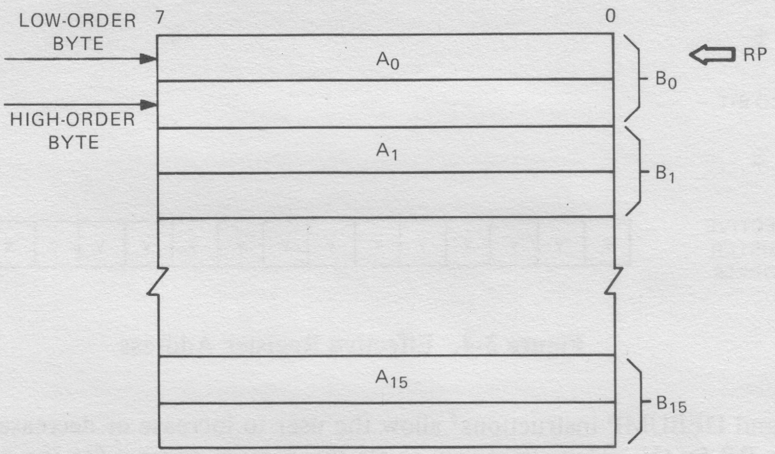


Figure 2-3. General Register Structure

Register Pointer

As indicated, the general registers can be located at any 8-byte boundary of the available memory space. One of the first actions in any program is to load the starting address of the general registers into the *Register Pointer* (RP). This 16-bit, on-chip register contains the address for register B0. The executing program later can provide a new set of registers for a subroutine, for example, merely by changing the contents of RP. If the original register set will be used later in the program, the current contents of RP can be saved (on the stack) before RP is altered. This capability makes it possible to save and restore the contents of a register set without actually moving data.

When an instruction accesses a general register, the processor computes the effective register address from the following values:

- The current contents of the RP; the three low-order bits must always be zero.

- An offset determined by the contents of the source or destination qualifier field (S or D field) in the instruction; the S field contains the number of the source register while the D field contains the destination register number.
- A processor-supplied bit (HI/LO bit) determined by whether the register being accessed is an A register (only the low-order byte) or a B register (the low-order byte followed by the high-order byte).

These quantities are aligned as shown in Figure 2-4 and added. Each value is treated as an unsigned integer.

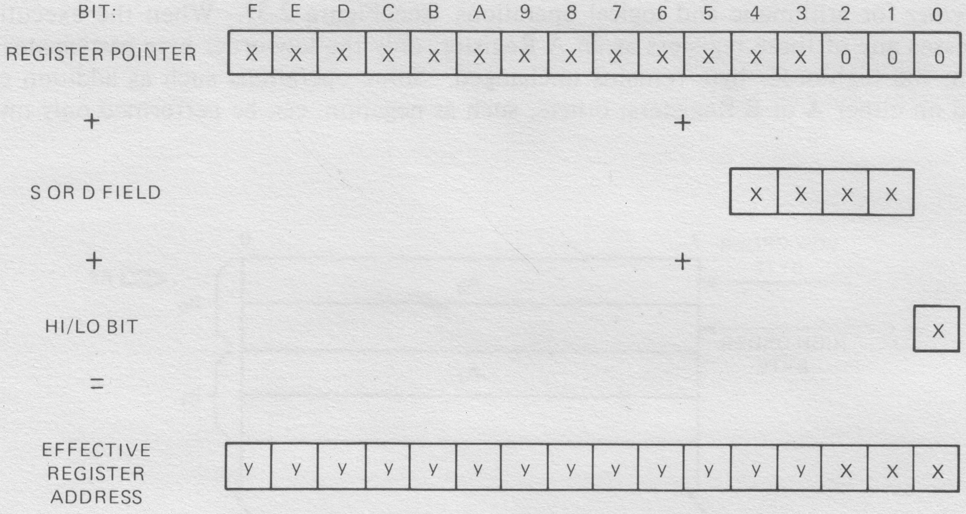

BIT: F E D C B A 9 8 7 6 5 4 3 2 1 0
REGISTER POINTER
X X X X X X X X X X X X X 0 0 0
+
S OR D FIELD
X X X X
+
HI/LO BIT
X
=
EFFECTIVE REGISTER ADDRESS
y y y y y y y y y y y y y X X X

Figure 2-4. Effective Register Address

The BUMP and DEBUMP instructions[3] allow the user to increase or decrease the content of bit 3 or 4 of the RP by '1'. This effectively shifts the starting address for the general registers up (increase) or down (decrease) in memory by 8 bytes (bit 3 changed) or 16 bytes (bit 4 changed). An 8-byte (or a 16-byte) shift can provide four (or eight) new 16-bit registers for a subroutine, for example, while preserving the contents of the original registers. The new register set partially overlaps the original set of registers whose contents can be restored upon completion of the subroutine by again altering the RP.

Condition Register

This 8-bit, on-chip register indicates the status of the program at any given point in its execution. The bits in the Condition Register (CR) describe the results of the most recent arithmetic or logical instructions and have the following meanings when set (i.e., the bit is a 1):

- Bit 0 — N, negative; high-order bit of result set
- Bit 1 — Z, zero result
- Bit 2 — V; arithmetic overflow from operation
- Bit 3 — C; carry (or borrow) out of the most-significant bit

---

[3] Also refer to **Special Register Instructions**, Chapter 3.

- Bit 4 — A; all ones in result
- Bit 5 — Odd; low-order bit of result set
- Bit 6 — Interrupt enabled
- Bit 7 — User-designated flag

Many MAC-8 instructions implicitly alter these condition bits (as can be seen from the descriptions of the instructions in Chapter 3).

The MAC-8 actually provides 16 conditions that can be logically tested (see **Conditions** in Chapter 3) with the conditional or decision-making instructions. The CR bits correspond to conditions 0 through 7, while conditions 8 through 15 are logical combinations of the CR bits.

Because the CR represents the state of the CPU, it is automatically saved (pushed) on the pushdown stack when an interrupt occurs. The CR is restored (popped from the stack) when the return from interrupt is executed.

Program Counter

A 16-bit on-chip register, the *Program Counter* (PC), contains the address of the instruction byte being executed or of the first byte for the next instruction. When the executing program calls a subroutine or is interrupted, the PC is automatically saved (using the pushdown stack) and set to the address of the first byte of the routine. It is restored (popped from the stack) when a return is executed.

Stack Pointer

Another 16-bit on-chip register, the *Stack Pointer* (SP), implements a pushdown stack or last-in, first-out list. The SP contains the memory address of the top of the stack, that is, the last item saved on the stack. It points to the *logical* top of the stack, which may not represent the highest absolute address allocated to the stack.

2.3 Stack

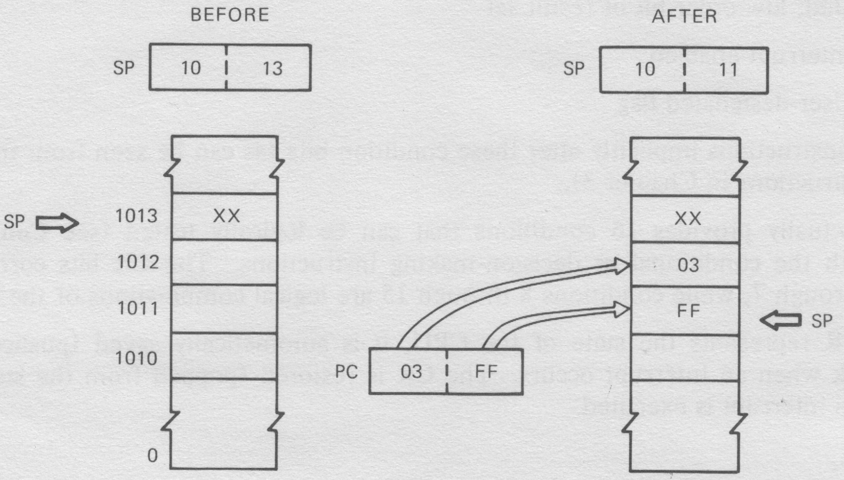
A programmer may find it convenient to incorporate subroutines in a program to modularize the software. The program also may include routines for handling interrupts caused by external devices. Both of these instances lead to the need to break the instruction sequence to execute the appropriate routine and then return to the main program at the point where it was interrupted.

The MAC-8 provides the capability for storing and restoring the PC, CR, and other registers (if needed) in a last-in, first-out manner. This facility, the pushdown stack, resides in RAM and is addressed using the SP. The SP identifies the location of the last item pushed onto the stack or the next item to be popped off the stack.

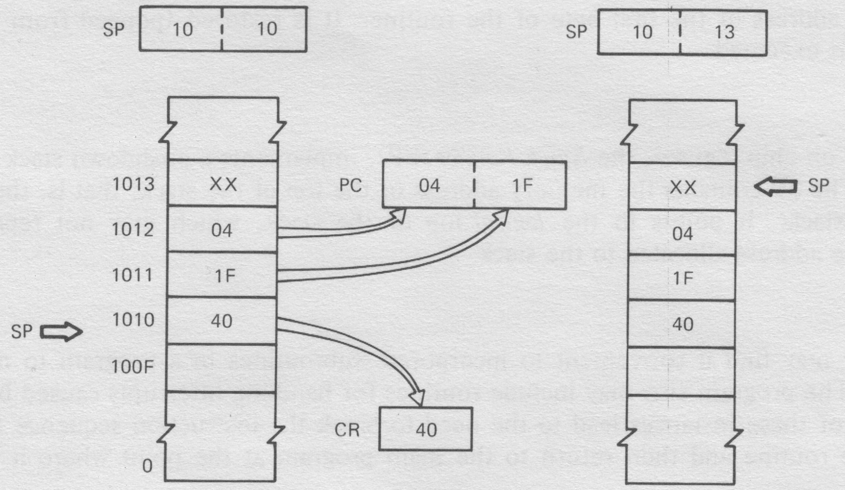
In the MAC-8, a push decrements and a pop increments the contents of the SP, hence the stack grows downward in memory. Only the item at the top of the stack (identified by the SP) and the items under it are valid.

Figure 2-5 illustrates a pushdown stack before and after a push and a pop occur. To push an item onto the stack, the SP is decremented by 1 or 2, depending on the length of the item, and the data are stored at that location. Conversely, for a pop the item is retrieved and the SP is incremented by 1 or 2, depending on the item's length.

The subroutine call (or the interrupt or reset acknowledge) is a push onto the stack, and a return is a pop from the stack. When executing the call (or acknowledge) and then the return, the CPU automatically uses the stack to save and restore the PC and, in the case of interrupts, the CR. The executing program may also use explicit push and pop instructions to place


BEFORE
AFTER
SP
10
13
SP
10
11
SP
1013
XX
1012
1011
1010
0
PC
03
FF
XX
03
FF
SP

A. Push PC


SP
10
10
SP
10
13
1013
XX
PC
04
1F
1012
04
1011
1F
SP
1010
40
100F
0
CR
40
XX
SP
04
1F
40

B. Pop PC and CR

Figure 2-5. Stack Manipulation

subroutine parameters and temporary variables on the stack. Special addressing modes (described in Chapter 3) allow easy access to items at or near the top of the stack.

The amount of memory space the user allocates to the stack is the only limit on the depth of nesting for subroutines and interrupts.

2.4 CPU Status

The Instruction Control Logic Array (ICLA) generates three status signals to indicate various internal states of the microprocessor. These signals are coded independently in the combinations identified in Table 2-1. The coded signals are applied to the three status pins (S_2,S_1,S_0) and can be used for monitoring the processor's status.

TABLE 2-1. STATUS ASSIGNMENTS

Function	Status Value (S_2,S_1,S_0)
PC Change	000
No Reportable Status	100
IR Fetch, no CR	001
Halt	101
RP Change	010
SP Change	110
IR Fetch, with CR	001
Trap or Trouble	111

In general, the status signal value indicates what will happen during the next clock cycle. This generalization does not apply, however, to the status indication with the CR; the CR is already on the data bus during the time that indication is true. The status values summarized in the table indicate the following event(s) will occur during the next clock cycle:

- *PC Change* — indicates discontinuity or new value entered in PC other than normal increment; new PC will appear on address bus.
- *No Reportable Status* — indicates no other signal is true.
- *IR Fetch, no CR* — indicates start of new in-line instruction at the next clock cycle; PC will appear on address bus; opcode will appear on data bus.
- *Halt* — indicates CPU has nothing to do; buses and pulse drivers are in high-impedance state.
- *RP Change* — indicates RP has been changed; new RP will appear on address bus.
- *SP Change* — indicates SP is changed; new SP will appear on address bus.
- *IR Fetch, with CR* — indicates start of new in-line instruction at the next clock cycle; PC will appear on address bus; CR is on data bus; opcode will appear on data bus next cycle.
- *Trap or Trouble* — indicates an illegal opcode, something went wrong internally, or chip has no power applied.

An Interrupt or a BRANCH, CALL, or RETURN instruction that changes the program flow results in a *PC Change* signal. This signal indicates that a new value, other than the normal increment, has been entered into the PC. If the processor started a non-READ cycle when this status signal was generated, an interrupt preamble was entered.

The *IR Fetch, no CR* signal generally indicates the instruction being executed does not affect the conditions. This is the case for the PUSH and POP instructions which save or restore registers using the stack. The CR also is not available on the bus for a SIGN EXTEND instruction.

The *RP Change* and *SP Change* signals frequently appear twice in an instruction cycle. Because the MAC-8 can process (i.e., read or write) only one byte in a clock cycle, two cycles are needed to LOAD the RP or SP with a 16-bit address or to PUSH or POP any 16-bit register. The first status signal indicates the CPU is processing the first byte and the second status indication occurs for the second byte. As a result, the final correct value is on the address bus only after the second status indication.

2.5 Interrupt, Trap, Reset

Certain exceptional events may occur that alter the course of the program being executed:

- *Interrupt* — an external device signals a request for service
- *Reset* — an external device resets the processor
- *Trap* — the CPU encounters an invalid opcode or some internal fault.

In the MAC-8, the first 256 bytes of memory are allocated to routines for responding to such events. Each event is associated with a fixed memory location to which control is transferred when the event occurs. Because these dedicated memory locations overlap (see Table 2-2), traps and resets can be simulated by appropriate interrupt signals or by direct BRANCH or CALL instructions from other routines.

TABLE 2-2. INTERRUPT ADDRESSES

Function	Address
Interrupt Acknowledge	X(FFFF)
Interrupt Vector	X(00yy)
Reset Vector	X(0000)
Trap Vector	X(0008)

where: X(0000) ⩽ X(00yy) ⩽ X(00FF)

Interrupt

An external device activates the $\overline{\text{INT REQ}}$ pin (pin 24) with a low logic level signal to request an interrupt. If the processor is executing a routine that should not be interrupted (i.e., CR bit 6 is 0), the interrupt is ignored (masked).

At the end of each instruction cycle when interrupts are enabled (i.e., CR bit 6 is 1), the CPU polls the $\overline{\text{INT REQ}}$ pin. If no signal is present, the CPU continues executing the program. However, if the pin is active, the CPU starts the interrupt preamble. The CR and PC are pushed on the stack and further interrupts are masked (i.e., CR bit 6 is cleared). The CPU then indicates an *Interrupt Acknowledge*. That is, it performs a normal READ operation addressing location X(FFFF). In most applications, this address does not represent regular memory; instead, the operation serves to acknowledge the interrupt request to the interrupting device. In response, the device supplies a data byte X(yy) which the processor uses to form the memory address X(00yy) for the appropriate interrupt vector. The value of the data byte identifies the interrupting device and acts as a pointer to the first instruction of the routine to process that type of interrupt.

Although the MAC-8 has only one interrupt pin, many interrupt procedures can be used to handle more than one interrupting source. A priority encoder chip (similar to the Intel[4] 8214 Priority Interrupt Controller described in Chapter 5, **Peripherals**) can be used to establish a different priority level for each interrupt. Furthermore, a daisy chain can be established in the program to process all interrupts at the same priority level, but in a predefined sequence. In daisy chaining, the CPU polls the possible interrupting devices to identify the source of the interrupt request. It passes an acknowledge to the first device in the chain; if this device did not request the interrupt, it passes the acknowledge to the next device; and so on until the requesting device identifies itself on the data bus. The device's position in the chain determines the priority level of its interrupt request.

---

[4] Registered trademark of Intel Corporation.

Reset

To reset the processor, an external device applies an active (low logic level) signal to the $\overline{\text{RESET}}$ pin (pin 25) and thereby generates a nonmaskable interrupt. The CPU responds as it does for an interrupt request, but ignores the data byte on the bus. Instead, the PC is cleared and program execution starts at location X(0000) which should contain the first instruction of the reset routine. The MAC-8 automatically initiates the reset sequence when V_{CC} power is first applied to the DIP assuming V_{DD} power is already present.

Trap

In the MAC-8, a trap occurs when the processor controller does not have a valid transition defined for the present state and input. This may happen when

- the processor attempts to execute an invalid opcode
- electrical transients disrupt the controller
- a fault develops in the controller.

Some transients and faults may not cause a trap, however. Traps, unlike interrupts, cannot be masked.

When the processor recognizes a trap condition (usually three clock cycles after the opcode fetch), it immediately stops executing the current instruction, pushes the CR and PC onto the stack, and clears the interrupt enable (i.e., sets CR bit 6 to 0). The PC then is set to location X(0008), which should contain the first instruction of the trap handler. Because the PC is automatically incremented after each memory fetch, the address of the interrupted instruction (which may have an invalid opcode) can be calculated from the saved PC.

2.6 Timing

Instruction length and time requirements depend upon the addressing mode selected rather than on the function being executed (see **Addressing Modes** in Chapter 3 and **Instruction Length** in Chapter 6). The opcode and address code contained in the first instruction byte indicate to the CPU how many memory bytes are used for the particular instruction. The address code further indicates whether addresses must be computed for the operands. In one clock (machine) cycle, the processor can either read a byte from memory or I/O, write a byte to memory or I/O, or perform an internal operation. Thus, the number of memory bytes to be fetched and the address computations required greatly influence the length of an instruction cycle. Memory access timing also has an affect.

The 16-bit address bus selects the appropriate memory location or I/O port from among the 64K locations of addressable space. An address will remain on the bus during any clock cycle in which an internal function occurs. Any device that recognizes this address should remain inactive until either a $\overline{\text{READ}}$ or a $\overline{\text{WRITE}}$ pulse is also present. The $\overline{\text{READ}}$, $\overline{\text{WRITE}}$, or DATA READY signal control data transfer via the 8-bit data bus; refer to **Access Timing** in Chapter 5.

Figures 2-6 and 2-7 illustrate the timing for a READ and a WRITE memory access for fast and slow memory, respectively. For both illustrations, +5V nominal (V_{CC}) is supplied to the on-chip TTL circuitry. Figure 2-6 shows the timing for fast memory accessing, i.e., for the 2-MHz version of the chip. The chip voltage V_{DD} is +12V nominal and the pulse rise and fall times are 1 ns/pF. Figure 2-7 illustrates the timing for slow memory accessing. In this case, V_{DD} can vary from +8.5V to +12V nominal and the pulse rise and fall times are 2.5 ns/pF. The DATA

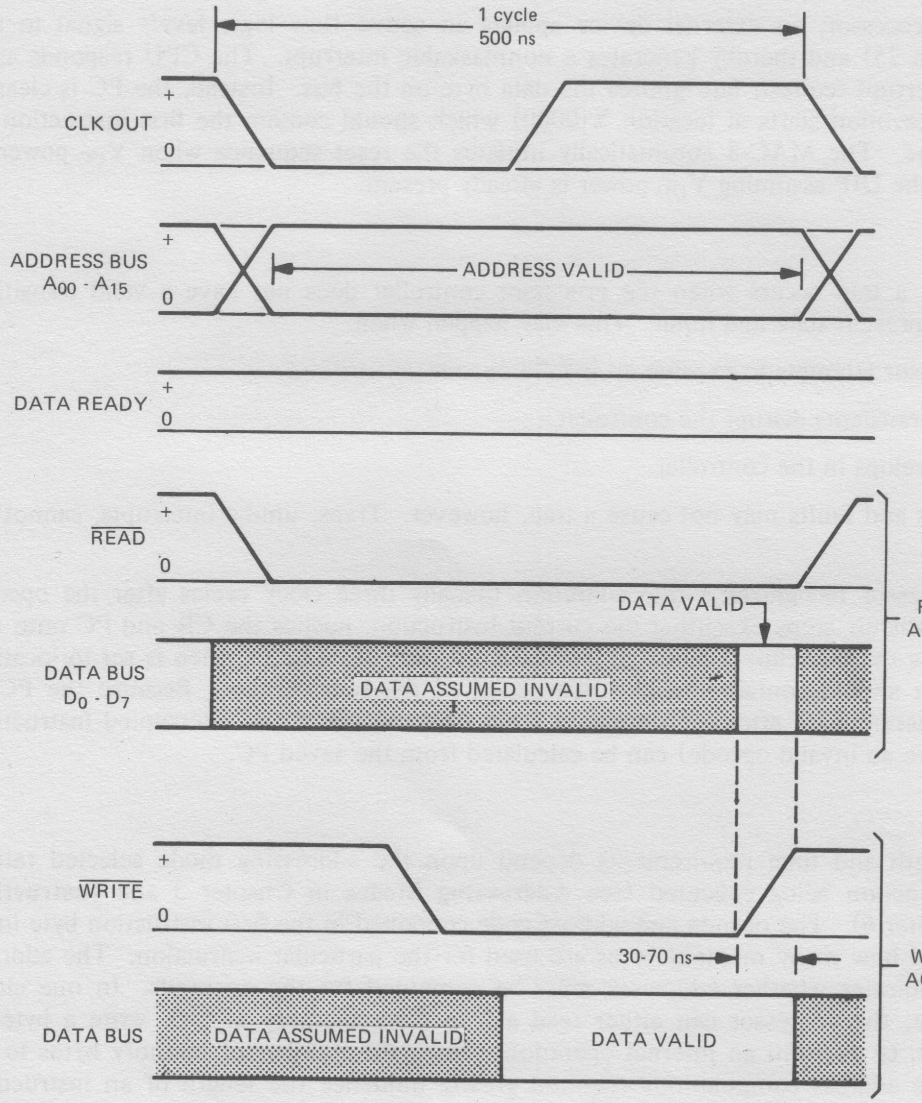

1 cycle
500 ns
CLK OUT
+
0
ADDRESS BUS
A00 - A15
ADDRESS VALID
DATA READY
READ
DATA VALID
READ ACCESS
DATA BUS
D0 - D7
DATA ASSUMED INVALID
WRITE
30-70 ns
WRITE ACCESS
DATA BUS
DATA ASSUMED INVALID
DATA VALID

Figure 2-6. Fast Memory Accessing

READY pulse level is significant only at strobe time.

Data Transfer

A low logic level signal on pin 22, $\overline{\text{READ}}$, activates a READ memory access. The $\overline{\text{READ}}$ pulse lasts until the end of the clock cycle and stays low until a non-READ cycle occurs.

At the end of the READ cycle, the information on the data bus is strobed into an internal latch. The voltage present on the data pin at the end of the data-valid window determines the latched binary level. Table 2-3 identifies these logic levels for fast memory accessing (Figure 2-6).

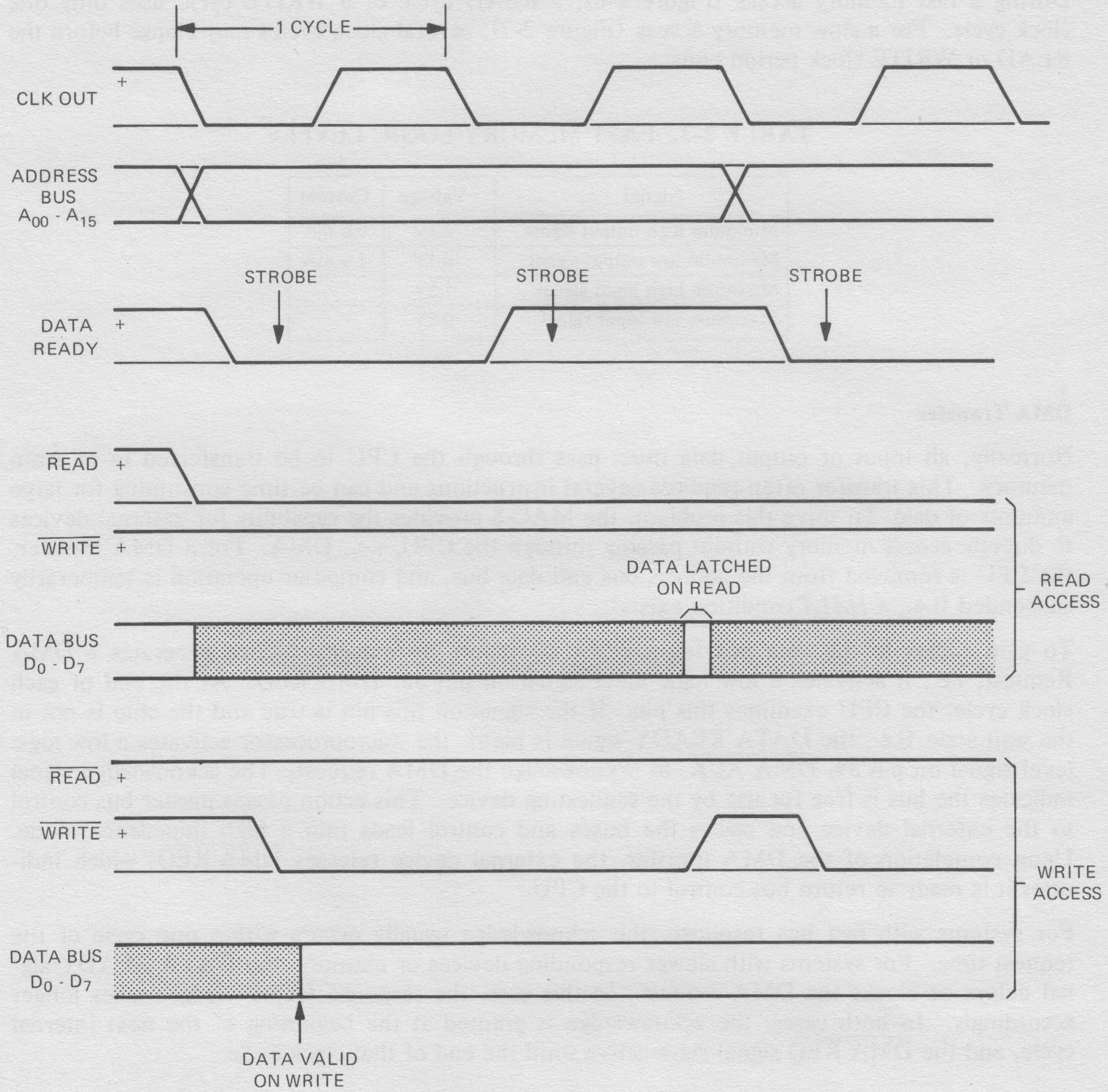

1 CYCLE
CLK OUT
+
ADDRESS BUS A_{00} - A_{15}
STROBE
STROBE
STROBE
DATA READY
+
READ +
WRITE +
DATA LATCHED ON READ
READ ACCESS
DATA BUS D_0 - D_7
READ +
WRITE +
WRITE ACCESS
DATA BUS D_0 - D_7
DATA VALID ON WRITE

Figure 2-7. Slow Memory Accessing

A low logic level signal on pin 23, $\overline{\text{WRITE}}$, activates a WRITE memory access. The $\overline{\text{WRITE}}$ pulse starts 150 ns (nominal) after the beginning of the WRITE clock period and terminates 5 ns before the cycle's end. The information on the data bus is valid at mid-cycle.

Figure 2-7 illustrates the timing for a READ or a WRITE cycle to accommodate slow memory accessing. The DATA READY input signal permits synchronizing of slow memory or I/O devices (as can be seen in Figure 2-7). This signal tells the CPU when to wait until the answering device is ready for data transfer on either a READ or a WRITE operation. The processor examines the DATA READY signal at mid-cycle. If the signal is true (high), the $\overline{\text{READ}}$ (or $\overline{\text{WRITE}}$) pulse terminates at the end of the cycle. If this signal is not true at mid-cycle, the chip enters a *wait* state and the $\overline{\text{READ}}$ (or $\overline{\text{WRITE}}$) signal remains low until after the next mid-cycle check.

During a fast memory access (Figure 2-6), a READ cycle or a WRITE cycle uses only one clock cycle. For a slow memory access (Figure 2-7), several clock cycles may elapse before the READ or WRITE clock period ends.

TABLE 2-3. FAST MEMORY LOGIC LEVELS

Signal	Voltage	Current
Minimum high output signal	2.8V	0.6 mA
Maximum low output signal	0.4V	1.6 mA
Minimum high input signal	1.8V	
Maximum low input signal	0.8V	

DMA Transfer

Normally, all input or output data must pass through the CPU to be transferred to or from memory. This transfer often requires several instructions and can be time-consuming for large amounts of data. To solve this problem, the MAC-8 provides the capability for external devices to directly access memory without passing through the CPU, i.e., DMA. For a DMA transfer, the CPU is removed from the address bus and data bus, and computer operation is temporarily suspended (i.e., a *HALT* condition exists).

To gain access to the data bus for a DMA operation, an external device generates a DMA Request; i.e., it activates a low logic level signal on pin 33, $\overline{\text{DMA REQ}}$. At the end of each clock cycle, the CPU examines this pin. If the signal on this pin is true and the chip is not in the wait state (i.e., the $\overline{\text{DATA READY}}$ signal is high), the microprocessor activates a low logic level signal on pin 34, $\overline{\text{DMA ACK}}$, to acknowledge the DMA request. The acknowledge signal indicates the bus is free for use by the requesting device. This action passes master bus control to the external device and places the buses and control leads into a high impedance mode. Upon completion of the DMA transfer, the external device releases $\overline{\text{DMA REQ}}$, which indicates it is ready to return bus control to the CPU.

For systems with fast bus response, the acknowledge usually occurs within one cycle of the request time. For systems with slower responding devices or memory, the DATA READY signal delays or blocks the DMA request. In this case, the response (acknowledge) takes longer accordingly. In both cases, the acknowledge is granted at the beginning of the next internal cycle, and the $\overline{\text{DMA REQ}}$ signal stays active until the end of that clock cycle.

Chapter 3

INSTRUCTION SET

3. INSTRUCTION SET

The MAC-8 instructions can be separated into the following major groups:

- *Arithmetic and Logical (A/L)* — instructions that perform an arithmetic or logical operation on one or two operands
- *Control Transfer* — instructions that alter the sequence of instruction execution
- *Special Register* — instructions that affect the contents of the processor's registers
- *Miscellaneous* — instructions that do not fit into any of the above groups.

This chapter briefly discusses MAC-8 programming and instruction format, and describes each of the instructions within the major groups. For additional details on the programming language and support software, refer to the *MAC-8 Programmer's Manual.*

3.1 Programming

Because the microprocessor is a machine designed to store and manipulate binary data, all of its instructions must either be written in or translated into binary (*machine language*) code. For this reason, each instruction is assigned a unique 8-bit binary number (opcode).

Short programs can be written easily in binary code. For programs containing hundreds of instructions, however, writing in binary code can be tedious and increases the possibility of error. In addition, such programs are difficult to debug.

The hexadecimal system of numbers, based on the digits 0 through 9 and the letters A through F, offers a partial solution to the problem. Because a one-to-one correspondence exists between the 16 hexadecimal digits and all possible combinations of 4 binary bits (see Table 3-1), a unique hexadecimal opcode[1] also exists for each MAC-8 instruction. Programs can be written in hexadecimal code and run using the MAC-8 system software.

Programs frequently contain many hundreds of lines of code, hence even hexadecimal code has its shortcomings. Such programs ideally should be coded in English-language statements, which are easily checked, debugged, and corrected. To support this need, the MAC-8 system includes software that allows programs to be coded in a C-like *assembly language.* This system software then translates the assembly language program into machine language code.

Thus, the integrated software system developed to support the MAC-8 allows programs to be coded in machine language, hexadecimal, or assembly language. The descriptions of the instructions in this chapter include all three of these forms.

3.1.1 Format

MAC-8 instructions vary in length from 1 to 6 bytes. The first instruction byte contains the opcode, hence it is generally called the *opcode byte.* For instructions that address memory, the opcode byte also contains 2 or 3 bits (called the *mode indicator*) that indicate how to form these

---

[1] Refer to the instruction descriptions in this chapter and to **Opcode Assignments** in Chapter 6.

TABLE 3-1. COMPARISON OF NUMBER SYSTEMS

Hexadecimal	Binary	Decimal
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
A	1010	10
B	1011	11
C	1100	12
D	1101	13
E	1110	14
F	1111	15

memory addresses; refer to the **Addressing Modes** discussion in this chapter.

In the A/L instruction group, the second instruction byte contains two 4-bit operand qualifiers — bits 4 through 7 comprise the *d field* which identifies the destination operand, while bits 0 through 3 comprise the *s field* which identifies the source operand. The single-operand or *monadic* instructions have one operand that serves as both source and destination, hence they only use the d field. The *dyadic* instructions have both a source and a destination operand, hence they use both operand qualifiers.

In the Control Transfer instruction group (e.g., some BRANCH and CALL instructions), the second instruction byte may contain two 4-bit fields. Here, bits 4 through 7 comprise the d field, which identifies the destination instruction. For the conditional form[2] of the instruction, bits 0 through 3 identify the condition to be tested.

Special Register and Miscellaneous instructions that address memory have the same basic format as the monadic or dyadic A/L instructions.

The number of instruction bytes remaining and their contents are determined by the addressing mode selected. If the instruction is to contain 16-bit data (or an address), the low-order byte of the data is stored first followed by the high-order byte at the next higher memory address. Any data required for the destination operand starts in the third instruction byte (or in the second byte for some Control Transfer instructions). Source operand data follows the destination operand data. If the addressing mode does not require destination operand data, the source operand data starts in the third instruction byte.

3.1.2 Addressing Modes

The addressing modes of an instruction are the different ways of forming the effective memory addresses for the operands of the instruction. Instructions that do not address memory do not have addressing modes. For instructions that have addressing modes, the first instruction byte contains 2 or 3 bits reserved for the mode indicator. Monadic instructions use a 2-bit mode indicator, while the dyadic instructions have a 3-bit mode indicator.

---

[2] Refer to **Conditions** and to the descriptions of these instructions in this chapter.

Three factors determine how an operand address is calculated:

1. Which mode indicator is used — generally modes 0 through 3 for monadic instructions and modes 0 through 7 for dyadic instructions.
2. Whether the operand is the source or the destination.
3. Which of the 16 registers is specified in the corresponding operand qualifier field. To extend the processor's addressing capability, special interpretations are used when the s or d field contains 15, i.e., register A15/B15.

Table 3-2 summarizes the eight basic addressing modes used when the s or d field specifies registers A0/B0 through A14/B14. The asterisk (*) indicates indirection, i.e., the contents of the register (or memory location) that follows the asterisk is a pointer to the data. Table 3-3 summarizes the extended addressing modes, i.e., for register A15/B15. For extended mode 2, the transfer instructions use the operand address identified by the brackets ([ ]) in this table.

TABLE 3-2. BASIC ADDRESSING MODES
(s/d field contains 0 — 14)

Mode	Source	Destination
0	Rs	Rd
1	Rs	*bd
2	Rs	*(bd+n)
3	Rs	*bd++
4	*(bs+n1)	*(bd+n2)
5	*bs	Rd
6	*(bs+n)	Rd
7	*bs++	Rd

TABLE 3-3. EXTENDED ADDRESSING MODES
(s/d field contains 15)

Mode	Source	Destination
0	*pc	R15
1	*pc	**pc
2	*pc	*(sp+n) [*(pc+n)]
3	*pc	*b15++
4	*(sp+n1)	*(sp+n2)
5	**pc	R15
6	*(sp+n)	R15
7	*b15++	R15

The notations used in the tables designate the following addressing modes:

- *Rs or Rd — Register —* the operand is the register *R* specified in the s or d field of the instruction. *R* designates a 16-bit B Register for 16-bit operations or an 8-bit A Register for 8-bit operations.

- **pc — Immediate —* the instruction contains the data for the source operand. The Program Counter (PC) points to a single data byte for 8-bit operations or to the low-order byte of a contiguous data-byte pair for 16-bit operations.
- ***pc — Direct —* the 16-bit address of the operand is specified in the instruction. The PC points to two contiguous bytes of the instruction that form the operand address. The first of these bytes contains the low-order byte of the address; the second contains the high-order byte. Here, the double asterisk designates a pointer to a pointer; i.e., the PC points to an address which is a pointer to the data.
- **bs or *bd — Indirect Memory —* the 16-bit B Register *b* specified in the s or d field contains the address of the operand.
- **(bs+n) or *(bd+n) — Offset Memory —* the 8-bit signed displacement *n* (stored in one instruction byte) is added to the contents of the specified B Register to determine the address of the operand.
- **(sp+n) — Offset Stack —* the 8-bit signed displacement *n* is added to the contents of the Stack Pointer (SP) to determine the address of the operand.
- **bs++ or *bd++ — Automatic Post Increment —* the contents of the specified B Register (which contains the address of the operand) are automatically incremented upon completion of the operation.
- **(pc+n) — Relative Transfer —* the 8-bit signed displacement *n* is added to the contents of PC[3] to determine the address of the destination operand. This is the starting address of the next instruction to be executed relative to the current PC. The new value (pc+n) replaces the current PC value, and program control transfers to that location.

3.1.3 Conditions

The MAC-8 processor provides a set of logical indicators that can be tested using the conditional instructions to control decision making and branching in the course of program execution. These instructions have one of two opcodes, one indicating the action to be taken if the condition is true and the other the action if it is false (see **Control Transfer Instructions** in this chapter and **Opcode Assignments** in Chapter 6). The 4-bit condition field identifies which of the 16 conditions is to be tested.

The first eight of the conditions (codes 0 through 7 in Table 3-4) are set or cleared by various MAC-8 instructions. The hexadecimal codes for these eight conditions correspond to their bit positions in the CR (condition 0 corresponds to bit 0, the least-significant bit). Conditions 8 through C (hexadecimal codes) are logical combinations of the CR bits. Because these bits are derived from the first eight conditions, they do not have to be stored.

The 16 conditions (summarized in Table 3-4) have the following functions:

0 *neg (N)* — This condition bit is set equal to the most-significant bit (MSB) of the result (the MAC-8 uses two's complement arithmetic). The condition is true (1) when the result is negative, i.e., the MSB is a 1.

1 *zero (Z)* — This condition bit is set true (1) when the result is all zeros. If an addition produces a zero result, the *zero* and the *carry* bits are set true.

---

[3] The PC is incremented after each memory fetch; hence, it contains the addresss of the opcode byte for the current instruction *plus* 2.

TABLE 3-4. CONDITIONS

Code	Name	Description
0	neg	N; negative
1	zero	Z; zero
2	ovfl	V; arithmetic overflow
3	carry	C; Carry (or borrow) out of MSB
4	ones	A; all ones
5	odd	Odd
6	enable	Interrupts enabled
7	flag	User flag
8	lt	N^V; arithmetically less than zero
9	lteq	(N^V)|Z; arithmetically less than or equal to zero
A	llteq	C|Z; logically less than or equal to zero
B	homog	A|Z; all ones or all zeros
C	shovfl	N^C; arithmetic left-shift overflow
D		Unused
E		Unused
F	always	Condition always true

2 *ovfl (V)* — This condition bit is set true when two's complement arithmetic produces an erroneous result. The true (1) condition indicates the result lies outside the valid range for two's complement numbers. The result of an arithmetic operation is incorrect when a carry occurs out of *either* the most-significant bit (MSB) or out of the bit adjacent to the MSB (that is, a carry out of or into the sign bit), but *not* both.

3 *carry (C)* — For ADD instructions with unsigned or two's complement signed numbers, this condition bit is set true (1) whenever there is a carry out of the MSB of the result. For SUBTRACT and COMPARE instructions, the two's complement of the source is added to the destination and the carry bit is set as in the ADD instructions. The carry bit is then complemented to give it the correct borrow indication for unsigned numbers.

4 *ones (A)* — This condition bit is set true when all bits in the result are ones.

5 *odd* — This condition bit is set equal to the least-significant bit (LSB) of the result. The result is an odd number only when the LSB is a 1.

6 *enable* — When this condition bit is set true, the MAC-8 will accept interrupts from peripheral devices. Otherwise, interrupts are masked.

7 *flag* — This condition bit provides a user-designated flag and is set true or false under program control.

8 *lt* — This condition is true if the result is arithmetically less than zero. For signed numbers, it is indicated by the logic operation N^V (i.e., the bitwise exclusive OR of condition bits 0 and 2). Recall that V is true when the result is in error because of arithmetic overflow, i.e., a carry occurred out of the sign bit *or* from the adjacent bit into the sign bit. Hence when V is true, the N condition gives an incorrect indication; that is, the carry out of or into the sign bit changed the sign bit. When V is false, the sign bit was not altered and N gives the correct indication.

9 *lteq* — This condition indicates the result is arithmetically less than or equal to zero. For signed numbers, the condition is indicated by the logic operation (N^V)|Z (i.e., the bitwise inclusive OR of N^V with Z, condition bit 1).

A *llteq* — This condition is true when the result is logically less than or equal to zero. It is applied only to unsigned numbers and is indicated by the logic operation C|Z (i.e., the bitwise inclusive OR of condition bits 3 and 1; C represents a borrow).

B *homog* — This condition is applied only to unsigned numbers and indicates the result is homogeneous, i.e., all zeros or all ones. It is indicated by the logic operation A|Z (i.e., the bitwise inclusive OR of condition bits 4 and 1).

C *shovfl* — This condition is true if, when shifting signed numbers to the left, the sign of the number changes. Unless the number being shifted is zero, the condition becomes true the first time the sign bit is different from the carry bit; hence, the condition is indicated by the logic operation N^C (i.e., the bitwise exclusive OR of condition bits 0 and 3).

D Not used at present.

E Not used at present.

F *always* — This *unconditional* condition satisfies a software need (that is, it provides unconditional instructions using the same coding format as the conditional instructions). When condition F (or always) is specified, a conditional instruction becomes unconditional and always takes the *branch*, i.e., it always alters the instruction sequence.

3.2 Description Notation

The remainder of this chapter describes the MAC-8 instructions within the following instruction groups: Arithmetic and Logical (A/L), Control Transfer, Special Register, and Miscellaneous. Each description consists of several parts: coded instruction, operation, addressing, instruction length, conditions, and examples.

3.2.1 Coded Instruction

The first part of the description presents the instruction coded in assembly language, binary, and hexadecimal.

Assembly Language

All alphabetic characters are written in lower case. The description illustrates the basic form of the instruction using the notation shown in Table 3-5 to identify registers.

The remaining notation consists of *tokens*, i.e., the lowest identifiable units of information: keywords, identifiers, constants, operators, and punctuation. Chapter 6 summarizes the various tokens; refer to the *MAC-8 Programmer's Manual* for additional details.

In many cases, the description illustrates only the basic address mode 0 form of the instruction. To determine the instruction's format for the other addressing modes, refer to the **Instruction Length** tables in Chapter 6; replace *op* with the appropriate operator.

TABLE 3-5. ASSEMBLY LANGUAGE NOTATION

Symbol	Description
ad	Destination register, a0 — a15
as	Source register, a0 — a15
bd	Destination register, b0 — b15
bs	Source register, b0 — b15
cr	Condition Register
pc	Program Counter
rp	Register Pointer
sp	Stack Pointer
m	16-bit address that must have been declared previously
N	Pseudo-condition; i.e., bit(s) to be tested.
n	Offset

Binary

The binary code illustrates the first two instruction bytes for instructions whose format varies according to the addressing mode selected. For other instructions, all instruction bytes are shown. The binary coding in all cases uses the notation shown in Table 3-6.

TABLE 3-6. BINARY NOTATION

Symbol	Description
xx	Mode identifier 0 to 3
xxx	Mode identifier 0 to 7
dddd	Destination operand qualifier
ssss	Source operand qualifier
cccc	Condition
nnnn	Bit(s) to be tested
offset	Signed 8-bit displacement
lo addr	Low-order byte of 16-bit address
hi addr	High-order byte of 16-bit address

Where other addressing modes are available for an instruction, only the basic address mode 0 form is illustrated. For the other addressing modes, destination operand data, if applicable, starts in instruction byte 3 followed by the source operand data. The low-order byte of 16-bit data or addresses is stored first followed by the high-order byte.

Hexadecimal

The descriptions illustrate the general format of the first two instruction bytes for instructions that have several addressing modes available. Because the mode selected alters the content of the first instruction (*opcode*) byte, the valid range is indicated. The mode selected also determines the number and content of remaining bytes for these instructions; refer to **Format**.

For other instructions, all instruction bytes are shown. The hexadecimal code uses the notation identified in Table 3-7.

TABLE 3-7. HEXADECIMAL NOTATION

Symbol	Description
d	Destination operand qualifier
s	Source operand qualifier
c	Condition
N	Bit(s) to be tested
offset	8-bit signed displacement
lo addr	Low-order byte of 16-bit address
hi addr	High-order byte of 16-bit address

3.2.2 Operation

First, a quasi-mathematical description of the operation is given. It borrows some of the assembly language notation for operators (refer to **Operators** in Chapter 6) but does not follow the lexical conventions. In particular, it uses the notation shown in Table 3-8.

TABLE 3-8. OPERATION NOTATION

Symbol	Description
*x	Indirection; thing pointed to by x
&x	Address of x
←	Result of operation to the right replaces contents of location at left
[!]	Not; indicates *false* condition; omit for *true* form of instruction
z<y,x>	Bits x (low order) through y (high order) of byte z (memory location or register)
⇄	Exchange contents of locations to right and left of arrows

The remainder of this part is a prose description of the operation.

3.2.3 Addressing

This part of the description identifies the valid addressing modes for the instruction. It is omitted for instructions that do not address memory.

3.2.4 Instruction Length

The instruction length is identified both in memory bytes and in machine cycles. The addressing mode selected determines the length of many instructions. Refer to the **Instruction Length** tables in Chapter 6 for the effect on bytes and cycles of each addressing mode.

3.2.5 Conditions

This part of the description identifies the conditions which are always altered by the operation and can be tested validly. Because the majority of the instructions at most affect the first six conditions (i.e., neg, zero, ovfl, carry, ones, and odd), only these conditions may be listed. A condition is not listed if the operation leaves it unaltered or if its final state is unpredictable.

Unless indicated otherwise, the conditions are altered according to the criteria stated previously in this chapter (see **Conditions**).

3.2.6 Examples

Representative examples of the instruction written in assembly language are provided. Because the examples are taken out of context, any declarations that may be required are not shown. Refer to the *MAC-8 Programmer's Manual* for details.

3.3 Arithmetic and Logical Instructions

These instructions perform some arithmetic or logical (A/L) operation on one or two operands, i.e., the contents of a register or a memory location or data provided in the instruction.

3.3.1 Monadic 8-bit Group

Monadic instructions have only one operand, called the destination. This group of monadic instructions operates on an 8-bit operand. Each instruction can be written in six different forms[4] representing the six different ways of addressing the operand using monadic address modes 0 through 3. The descriptions illustrate the general format of the instruction using basic monadic address mode 0.

Increment

++*ad;*

0010	10xx
dddd	0000

Hex	28	to	2B
	d0		d0

Operation

destination ← destination + 1

Increments the 8-bit contents of the destination operand by 1. The data is treated as an unsigned number.

Addressing:	All 8-bit dyadic modes
Bytes:	2 to 4
Cycles:	7 to 14
Conditions:	neg, zero, carry, ones, odd
Examples:	++a3; ++thing;

---

[4] The special interpretation applied to register 15 for modes 1 and 2 enables four modes to provide six instruction formats; refer to **Instruction Length, Monadic Instructions** in Chapter 6.

Decrement

− −ad;

0010	10xx
dddd	1000

Hex 28 to 2B
d8 d8

Operation

destination ← destination − 1

Decrements the 8-bit contents of the destination operand by 1. The data is treated as an unsigned number.

Addressing: All 8-bit monadic modes

Bytes: 2 to 4

Cycles: 7 to 14

Conditions: neg, zero, carry, ones, odd

Examples: −−al;
−−thing;

Complement

ad = ¬ad;

0010	11xx
dddd	0000

Hex 2C to 2F
d0 d0

Operation

destination ← one's complement of destination

Complements the 8-bit contents of the destination operand, i.e., changes all zeros to ones and all ones to zeros. The ASCII tilde (~) and EBCDIC hook (¬) may be interchanged, as needed, when coding the instruction.

Addressing: All 8-bit monadic modes

Bytes: 2 to 4

Cycles: 7 to 14

Conditions: neg, zero, ones, odd

Examples: a6 =¬a6;

Negate

ad = −ad;

0010	01xx
dddd	0000

Hex 24 to 27
d0 d0

Operation

destination ← two's complement of destination

Negates the 8-bit contents of the destination operand, i.e., changes zeros to ones and ones to zeros (as for the one's complement), and then adds one to the result to obtain the two's complement. Attempting to negate the largest negative number (i.e., hex 80 or decimal −128) leaves the destination unchanged and sets the *ovfl* condition.

Addressing: All 8-bit monadic modes

Bytes: 2 to 4

Cycles: 7 to 14

Conditions: neg, zero, ovfl, carry, ones, odd

Example: a0 = −a0;

Zero

ad = 0;

0010	00xx
dddd	0000

Hex 20 to 23
d0 d0

Operation

destination ← 0

The 8-bit contents of the destination operand are set to zero.

Addressing: All 8-bit monadic modes

Bytes: 2 to 4

Cycles: 7 to 14

Conditions: neg, zero, ones, odd

Examples: a5 =0;
thing =0;

Rotate Left

ad = ad <<< 1;

0011	01xx
dddd	0001

Hex 34 to 37
d1 d1

Operation

destination ← destination rotated left 1 bit

Rotates the 8-bit contents of the destination operand to the left by one bit. The bit rotated off the left end of the destination appears at the other end and simultaneously becomes the carry condition.

Addressing: 8-bit dyadic modes 0 through 3

Bytes: 2 to 5

Cycles: 9 to 16

Conditions: neg, zero, carry, ones, odd

Example: a9 =a9 <<< 1;

Rotate Right

ad = ad >>> 1;

0011	01xx
dddd	1111

Hex 34 to 37
dF dF

Operation

destination ← destination rotated right 1 bit

Rotates the 8-bit contents of the destination operand to the right by one bit. The bit rotated off the right end of the destination appears at the other end and simultaneously becomes the carry condition.

Addressing: 8-bit dyadic modes 0 through 3

Bytes: 2 to 5

Cycles: 9 to 16

Conditions: neg, zero, carry, ones, odd

Example: a6 = a6 >>> 1;

Rotate 9 Left

ad = ad $<< *1;*

0011	11xx
dddd	0001

Hex 3C to 3F
dl dl

Operation

destination ← destination and carry rotated left 1 bit

Concatenates the 8-bit contents of the destination operand with the carry condition to form a 9-bit quantity. All 9 bits then are rotated to the left by one bit. The bit rotated off the left end appears at the other end. The final conditions settings are based on the 8-bit portion of the destination.

Addressing: 8-bit dyadic modes 0 through 3

Bytes: 2 to 5

Cycles: 9 to 16

Conditions: neg, zero, carry, ones, odd

Example: a14 = a14 $<< 1;

Rotate 9 Right

ad = ad >>$ *1;*

0011	11xx
dddd	1111

Hex 3C to 3F
dF dF

Operation

destination ← destination and carry rotated right 1 bit

Concatenates the 8-bit contents of the destination operand with the carry condition to form a 9-bit quantity. All 9 bits are rotated to the right by one bit. The bit rotated off the right end appears at the other end. The final conditions settings are based on the 8-bit portion of the destination.

Addressing:	8-bit dyadic modes 0 through 3
Bytes:	2 to 5
Cycles:	9 to 16
Conditions:	neg, zero, carry, ones, odd
Example:	a3 =a3 >>$ 1;

Shift Left Arithmetic

*ad = ad * 2;*

0011	00xx
dddd	0001

Hex 30 to 33
d1 d1

Operation

destination ← destination shifted left arithmetically 1 bit

Shifts the 8-bit contents of the destination operand to the left by one bit. The bit shifted out becomes the carry condition, and the position vacated is filled with 0. If a non-zero destination is shifted left enough times, its sign will be changed, which will set the *ovfl* condition. This occurrence can be detected by testing the *shovfl* condition.

Addressing:	8-bit dyadic modes 0 through 3
Bytes:	2 to 5
Cycles:	9 to 16
Conditions:	neg, zero, carry, ones, odd

Examples: a9 = a9 * 2;
thing = thing * 2;

Shift Right Arithmetic

ad = ad / 2;

0011	00xx
dddd	1111

Hex 30 to 33
dF dF

Operation

destination ← destination shifted right arithmetically 1 bit

Shifts the 8-bit contents of the destination operand to the right by one bit. The bit shifted out becomes the carry condition, and the vacated position is filled with itself; the sign (two's complement) of the operand is preserved. Overflow cannot occur on a right shift.

Addressing:	8-bit dyadic modes 0 through 3
Bytes:	2 to 5
Cycles:	9 to 16
Conditions:	neg, zero, carry, ones, odd
Examples:	a9 = a9 / 2; thing = thing / 2;

Shift Left Logical

ad = ad << 1;

0011	10xx
dddd	0001

Hex 38 to 3B
d1 d1

Operation

destination ← destination shifted left logically 1 bit

Shifts the 8-bit contents of the destination operand to the left by one bit. The bit shifted out becomes the carry condition, and the vacated position is filled with zero.

Addressing: 8-bit dyadic modes 0 through 3

Bytes: 2 to 5

Cycles: 9 to 16

Conditions: neg, zero, carry, ones, odd

Examples: a9 = a9 << 1;
thing = thing << 1;

Shift Right Logical

ad = ad >> 1;

0011	10xx
dddd	1111

Hex 38 to 3B
dF dF

Operation

destination ← destination shifted right logically 1 bit

Shifts the 8-bit contents of the destination operand to the right by one bit. The bit shifted out appears as the carry condition, and the vacated position is filled with zero.

Addressing: 8-bit dyadic modes 0 through 3

Bytes: 2 to 5

Cycles: 9 to 16

Conditions: neg, zero, carry, ones, odd

Examples: a9 = a9 >> 1;
thing = thing >> 1;

3.3.2 Monadic 16-bit Group

These monadic instructions operate on a 16-bit operand. The instructions can only be written in the form illustrated, i.e., 16-bit monadic address mode 0.

Increment 16

++*bd;*

0110	1000
dddd	0000

Hex 68
d0

Operation

destination ← destination + 1

Increments the 16-bit contents of the destination operand by 1. The data is treated as an unsigned number.

Addressing:	Monadic mode 0
Bytes:	2
Cycles:	12
Conditions:	neg, zero, carry
Example:	++b3;

Decrement 16

--*bd;*

0110	1000
dddd	1000

Hex 68
d8

Operation

destination ← destination − 1

Decrements the 16-bit contents of the destination operand by 1. The data is treated as an unsigned number.

Addressing:	Monadic mode 0
Bytes:	2
Cycles:	12
Conditions:	neg, zero, carry
Example:	--b1;

Zero 16

bd = 0;

0110	0000
dddd	0000

Hex 60
d0

Operation

destination ← 0

Sets the 16-bit contents of the destination operand to zero.

Addressing:	Monadic mode 0
Bytes:	2
Cycles:	12
Conditions:	neg, zero, ones, odd
Example:	b3 =0;

3.3.3 Dyadic 8-bit Group

Dyadic instructions have two operands — the source (s) and the destination (d); the result of the operation is stored back in the destination, overwriting its original value. The 8-bit dyadic instructions operate on two 8-bit operands. Each instruction can be written in 21 different forms[5] representing the 21 ways of addressing the operands using all of the dyadic address modes. The descriptions illustrate the instruction format for dyadic address mode 0.

> **NOTE:** For mode 4, these instructions use the mode 4 opcode for the corresponding 16-bit instructions.

Add

ad = ad + as;

1010	1xxx
dddd	ssss

Hex A8 to AB, E4 (mode 4), AD to AF
ds ds ds ds ds

opcode AC not used

Operation

destination ← destination + source

Adds the 8-bit contents of the source operand to the destination operand, using two's-complement arithmetic, and places the result in the destination. The source is not altered unless the source and destination operands are the same.

---

[5] Refer to **Instruction Length, Dyadic 8-Bit Instructions** in Chapter 6.

Addressing: All 8-bit dyadic modes

Bytes: 2 to 5

Cycles: 9 to 22

Conditions: neg, zero, ovfl, carry, ones, odd

Examples: a4 = a4 + 2;
thing = thing + a7;

Compare

ad − as;

1011	0xxx
dddd	ssss

Hex	B0	to	B3,	F4 (mode 4),	B5	to	B7
	ds		ds	ds	ds		ds

opcode B4 not used

Operation

(destination − source) sets CR

Subtracts the 8-bit contents of the source operand from the destination operand, using two's-complement arithmetic, and sets the condition bits according to the result. The source and the destination are not altered.

Addressing: All 8-bit dyadic modes

Bytes: 2 to 5

Cycles: 9 to 22

Conditions: neg, zero, ovfl, carry, ones, odd

Examples: a6 − a7;
a6 − *(b7 + 2);

Move

ad = as;

1000	0xxx
dddd	ssss

Hex	80	to	83,	C4	(mode 4),	85	to	87
	ds		ds	ds		ds		ds

opcode 84 not used

Operation

destination ← source

Replaces the destination operand with the 8-bit contents of the source operand. The source is not altered.

Addressing:	All 8-bit dyadic modes
Bytes:	2 to 5
Cycles:	9 to 22
Conditions:	neg, zero, ones, odd
Examples:	a8 = a7; thing = a8;

Subtract

ad = ad − as;

1010	0xxx
dddd	ssss

Hex	A0	to	A3,	E4	(mode 4),	A5	to	A7
	ds		ds	ds		ds		ds

opcode A4 not used

Operation

destination ← destination − source

Subtracts the 8-bit contents of the source operand from the destination operand using two's-complement arithmetic, and places the results in the destination. The source is unchanged unless the source and destination operands are the same, in which case the result is zero.

Addressing:	All 8-bit dyadic modes
Bytes:	2 to 5
Cycles:	9 to 22
Conditions:	neg, zero, ovfl, carry, ones, odd
Example:	a2 = a2 − a6;

Test

test(ad,as);

1011	1xxx
dddd	ssss

Hex	B8	to	BB,	FC (mode 4),	BD	to	BF
	ds		ds	ds	ds		ds

opcode BC not used

Operation

destination masked by source sets CR

Performs a bit-by-bit AND operation between the 8-bit source and destination operands, leaving the source and destination unchanged, and uses the result to set the conditions. However, the *ones* condition is set differently. That is, if the destination has a 1 in every bit in which the source has a 1, the *ones* condition is set true (1); otherwise, the *ones* condition is set false (0).

Addressing:	All 8-bit dyadic modes
Bytes:	2 to 5
Cycles:	9 to 22
Conditions:	neg, zero, ones, odd
Example:	test(memloc,mask);

AND

ad = ad & as;

1001	1xxx
dddd	ssss

Hex	98	to	9B,	DC (mode 4),	9D	to	9F
	ds		ds	ds	ds		ds

opcode 9C not used

Operation

destination ← destination ANDed with source

Performs the bit-by-bit logical AND between the 8-bit contents of the source and destination operands and places the result in the destination. If the corresponding bits in the source and the destination are both 1, the resulting bit is 1; otherwise, the resulting bit is 0. The source is unchanged.

Addressing: All 8-bit dyadic modes

Bytes: 2 to 5

Cycles: 9 to 22

Conditions: neg, zero, ones, odd

Example: a14 = a14 & ramloc;

EXCLUSIVE OR (XOR)

ad = ad^ as;

1000	1xxx
dddd	ssss

Hex	88 to 8B,	CC (mode 4),	8D to 8F
	ds ds	ds	ds ds

opcode 8C not used

Operation

destination ← destination XORed with source

Performs the bit-by-bit logical XOR between the 8-bit contents of the source and destination operands and places the result in the destination. If either but not both of the corresponding bits in the source and the destination is 1, the resulting bit is 1; otherwise, the resulting bit is 0. The source is unchanged unless the source and destination operands are the same, in which case the result is zero.

Addressing: All 8-bit dyadic modes

Bytes: 2 to 5

Cycles: 9 to 22

Conditions: neg, zero, ones, odd

Example: loc = loc ^ a8;

OR

ad = ad | as;

1001	0xxx
dddd	dddd

Hex 90 to 93, D4 (mode 4), 95 to 97
ds ds ds ds ds

opcode 94 not used

Operation

destination ← destination ORed with source

Performs the bit-by-bit logical OR between the 8-bit contents of the source and destination operands and places the result in the destination. If either or both of the corresponding bits in the source and the destination are 1, the resulting bit is 1; otherwise, the resulting bit is 0. The source is unchanged.

Addressing:	All 8-bit dyadic modes
Bytes:	2 to 5
Cycles:	9 to 22
Conditions:	neg, zero, ones, odd
Examples:	ramloc = ramloc | a0; a0 = a0 | ramloc;

3.3.4 Dyadic 16-bit Instructions

The 16-bit dyadic instructions operate on two 16-bit operands. Each instruction can be written in 17 different forms[6] representing the different ways of addressing the operands using dyadic address modes 0 through 3 and 5 through 7.

Add 16

bd = bd + bs;

1110	1xxx
dddd	ssss

Hex E8 to EB, ED to EF
ds ds ds ds

Operation

destination = destination + source

Adds the 16-bit contents of the source operand to the destination operand, using two's-complement arithmetic, and places the result in the destination. The source is not altered.

Addressing:	16-bit dyadic modes
Bytes:	2 to 6

---

6. Refer to **Instruction Length, Dyadic 16-Bit Instructions** in Chapter 6.

Cycles: 14 to 22

Conditions: neg, zero, ovfl, carry

Examples: b2 = b2 + b1;
*b5++ = *b5++ + 9;

Compare 16

bd − bs;

1111	0xxx
dddd	ssss

Hex	F0	to	F3,	F5	to	F7
	ds		ds	ds		ds

Operation

(destination − source) sets CR

Subtracts the 16-bit contents of the source operand from the destination operand, using two's-complement arithmetic, and sets the condition bits according to the result. The source and the destination are not altered.

Addressing: All 16-bit dyadic modes

Bytes: 2 to 6

Cycles: 14 to 22

Conditions: neg, zero, ovfl, carry

Example: thing − b3;

Move 16

bd = bs;

1100	0xxx
dddd	ssss

Hex	C0	to	C3,	C5	to	C7
	ds		ds	ds		ds

Operation

destination ← source

Copies the 16-bit contents of the source operand into the destination operand. The source is unchanged.

Addressing: All 16-bit dyadic modes

Bytes: 2 to 6

Cycles: 14 to 22

Conditions: neg, zero

Examples: b0 = b7;
thing = b8;

Subtract 16

bd = bd − bs;

1110	0xxx
dddd	ssss

Hex	E0	to	E3,	E5	to	E7
	ds		ds	ds		ds

Operation

destination ← destination − source

Subtracts the 16-bit contents of the source operand from the destination, using two's-complement arithmetic, and places the result in the destination. The source is unchanged unless the source and destination operands are the same, in which case the result is zero.

Addressing: All 16-bit dyadic modes

Bytes: 2 to 6

Cycles: 14 to 22

Conditions: neg, zero, ovfl, carry

Example: b12 = b12 − thing;

AND 16

bd = bd & bs;

1101	1xxx
dddd	ssss

Hex	D8	to	DB,	DD	to	DF
	ds		ds	ds		ds

Operation

destination ← destination ANDed with source

Performs the bit-by-bit logical AND between the 16-bit contents of the source and destination operands, and places the result in the destination. If the corresponding bits in the source and destination are both 1, the resulting bit is 1; otherwise, the resulting bit is 0. The source is unchanged.

Addressing: All 16-bit dyadic modes

Bytes: 2 to 16

Cycles: 14 to 22

Conditions: neg, zero

Example: *(b5+6) =*(b5+6) & b8;

EXCLUSIVE OR 16 (XOR 16)

bd = bd^ bs;

1100	1xxx
dddd	ssss

Hex	C8	to	CB,	CD	to	CF
	ds		ds	ds		ds

Operation

destination ← destination XORed with source

Performs the bit-by-bit logical XOR between the 16-bit contents of the source and destination operands and places the result in the destination. If either but not both of the corresponding bits in the source and destination is 1, the resulting bit is 1; otherwise, the resulting bit is 0. The source is unchanged unless both operands are the same, in which case the result is zero.

Addressing: All 16-bit dyadic modes

Bytes: 2 to 6

Cycles: 14 to 22

Conditions: neg, zero

Example: b5 =b5 ^ ones;

OR 16

_bd = bd | bs;_

1101	0xxx
dddd	ssss

Hex D0 to D3, D5 to D7
ds ds ds ds

Operation

destination ← destination ORed with source

Performs the bit-by-bit logical OR between the 16-bit contents of the source and destination operands and places the result in the destination. If either or both of the corresponding bits in the source and destination are 1, the resulting bit is 1; otherwise, the resulting bit is 0. The source is unchanged.

Addressing: All 16-bit dyadic modes

Bytes: 2 to 6

Cycles: 14 to 22

Conditions: neg, zero

Examples: b3 = b3 | thing;
thing = thing | b3;

3.4 Control Transfer Instructions

The MAC-8 normally executes instructions in sequential order. The control transfer instructions can change the contents of the Program Counter, thus altering the instruction sequence.

3.4.1 Call Subroutine Instructions

The call-subroutine instructions save the current contents of the Program Counter before transferring control to the specified subroutines. This enables the processor to return to the normal instruction sequence when that routine is completed. In the unconditional form, the call to the subroutine is always executed. The conditional (T/F) form first tests the state of the Condition Register against a specified condition. If the condition is satisfied, the call is executed; otherwise, the processor executes the next sequential instruction.

Call

_*m();_

or

symbol();

0111	1001
low addr	
hi addr	

Hex 79
lo addr
hi addr

Operation

SP ← SP − 2; *SP ← updated PC; PC ← destination address

Decrements the Stack Pointer by 2 and pushes the address of the next instruction following the call onto the stack. Places the address of the subroutine's first instruction into the Program Counter. The destination is addressed as mode 1 where the d field contains 15, but the s/d byte is omitted.

Addressing:	Abbreviated monadic mode 1
Bytes:	3
Cycles:	11
Conditions:	Unchanged
Example:	subr();

Call on Condition (T/F)

True -

if (condition) symbol();

0110	1001
dddd	cccc

Hex 69
dc

False -

if (!condition) symbol();

0110	0001
dddd	cccc

Hex 61
dc

Operation

```
If [!]condition then
    SP ← SP − 2; *SP ← updated PC; PC ← destination address
```

If the indicated condition is satisfied or when the condition specified is F (i.e., *always*), decrements the Stack Pointer, pushes the address of the next instruction after the call onto the stack, and places the address of the subroutine's first instruction in the Program Counter. Otherwise, continues execution with the next sequential instruction.

Addressing:	Monadic mode 1
Bytes:	2 to 4
Cycles:	11 to 12, if condition is met; otherwise, 6 to 7
Conditions:	Unchanged
Examples:	if (lt) subr (); if (!shovfl) subr ();

3.4.2 Return Instructions

These instructions return execution control to a routine upon completion of another routine. When executed, a return restores the processor to the state it had prior to transferring execution to that routine. Returns from subroutines can be conditional or unconditional. In the unconditional form, the return is always executed. The conditional (T/F) form first tests the Condition Register against a specified condition. If the condition is satisfied, the return is executed; otherwise, execution continues with the next instruction. Returns from interrupts are always unconditional.

These instructions do not address memory, hence they do not have address modes.

Return From Interrupt

ireturn();

0110	0111

Hex 67

Operation

```
PC ← *SP; SP ← SP + 2; CR ← *SP; SP ← SP + 1
```

Returns to the main program from a routine that services an interrupt. Pops the 16-bit return address from the top of the stack, placing the first byte in the low-order byte and the second byte in the high-order byte of the Program Counter. Pops the next byte from the stack and places it in the Condition Register.

Bytes: 1

Cycles: 9

Conditions: All; determined by byte popped from the stack

Return From Subroutine

return;

0110	0110

Hex 66

Operation

PC ← *SP; SP ← SP +2

Pops the 16-bit return address from the top of the stack and places this address into the Program Counter. The first byte popped becomes the low-order byte of the Program Counter. Program execution resumes with the next instruction following the call to this subroutine.

Bytes: 1

Cycles: 8

Conditions: Unchanged

Return From Subroutine on Condition (T/F)

True -

if (condition) return;

0110	0100
0000	cccc

Hex 64
0c

False -

if (!condition) return;

0110	0101
0000	cccc

Hex 65
0c

Operation

```
If [!]condition then
        PC ← *SP; SP ← SP +2
```

If the indicated condition is satisfied or when the condition specified is F (i.e., *always*), pops the 16-bit return address from the top of the stack and places this address into the Program Counter; the first byte popped becomes the low-order byte of the Program Counter. Otherwise, execution continues with the instruction following the conditional return.

Bytes: 2

Cycles: 8, if condition is met; otherwise, 7

Conditions: Unchanged

Examples: if (flag) return;
if (!odd) return;

3.4.3 Branch Instructions

These instructions allow the next instruction to be fetched from any location in memory. Unlike the Call instructions, they do not save the Program Counter. Execution of a branch instruction can be conditional or unconditional. The unconditional form is always executed, i.e., program execution transfers to the specified instruction. The conditional (T/F) form first tests the state of the Condition Register against a specified condition. If the condition is satisfied, execution then transfers to the specified instruction; otherwise, the next sequential instruction is executed.

Branch 1

*goto *m;*

or

goto symbol;

0101	1001
lo addr	
hi addr	

Hex 59
lo addr
hi addr

Operation

PC ← destination address

Loads the destination address supplied in the instruction into the Program Counter. Program execution transfers to that memory address. The destination is addressed as mode 1 where the d field contains 15, but the s/d byte is omitted.

Addressing: Abbreviated monadic mode 1

Bytes: 3

Cycles: 6

Conditions: Unchanged

Example: goto label;

Branch 2

goto *(pc + n);

0101	1000
offset	

Hex 58
offset

Operation

PC ← destination address

Adds the 8-bit signed displacement to the contents of the Program Counter to determine the destination address, and places this address into the Program Counter. Program execution transfers to that location. The destination is addressed as mode 2 where the d field contains 15, but the s/d byte is omitted.

Addressing: Abbreviated monadic mode 2

Bytes: 2

Cycles: 5

Conditions: Unchanged

Example: goto *(pc +12);

Branch 1 on Condition (T/F)

True -

if (condition) goto *bd;

0100	1001
dddd	cccc

Hex 49
dc

False -

if (!condition) goto *bd;

0100	0001
dddd	cccc

Hex 41
dc

Operation

If [!]condition then PC ← destination address

If the indicated condition is satisfied or if the condition is F (_always_), places the destination address in the Program Counter. Otherwise, executes the next sequential instruction.

Addressing: Monadic mode 1

Bytes: 2 or 4

Cycles: 6 or 7

Conditions: Unchanged

Examples: if (zero) goto label;
if (!ones) goto *b2;

Branch 2 on Condition (T/F)

True -

if (condition) goto *(bd + n);

0100	1000
dddd	cccc
offset	

Hex 48
dc
offset

False -

if (!condition) goto *(bd + n);

0100	0000
dddd	cccc
offset	

Hex 40
dc
offset

If the indicated condition is satisfied or if the condition is F (*always*), adds the 8-bit signed displacement to the designated B Register (or to the Program Counter) to determine the destination address; places this address in the Program Counter. Otherwise, executes the next sequential instruction.

Addressing: Monadic mode 2, but if d field contains 15, destination address is PC + n

Bytes: 3

Cycles: 8, if condition is met; otherwise, 6

Conditions: Unchanged

Examples: if (zero) goto label;
if (!ones) goto label;

Branch on Register Bit (T/F)

True -

if (bit(N,as)) goto symbol;

0101	1010
ssss	nnnn
offset	

Hex 5A
sN
offset

False -

if (!bit(N,as)) goto symbol;

0101	0010
ssss	nnnn
offset	

Hex 52
sN
offset

Operation

If [!]bit(N,source) then PC ← destination address

Tests the 8-bit contents of register *as* according to the pseudo-condition N. If the indicated condition is satisfied or if N is 15 (*always*), adds the 8-bit signed displacement to the contents of the Program Counter to determine the destination address, and places this address in the Program Counter. Otherwise, executes the next sequential instruction.

Addressing:	Source, mode 2; Destination, abbreviated mode 2
Bytes:	3
Cycles:	8, if condition is met; otherwise, 7
Conditions:	Unchanged
Examples:	if (bit(4,a6)) goto dest; if (!bit(4,a1)) goto dest;

Branch on Memory Bit (T/F)

True -

*if (bit(N,*bs)) goto symbol;*

0101	1011
ssss	nnnn
offset	

Hex 5B
sN
offset

False -

*if (!bit(N,*bs)) goto symbol;*

0101	0011
ssss	nnnn
offset	

Hex 53
sN
offset

Operation

If [!]bit(N,source) then PC ← destination address

Tests the 8-bit contents of memory location **bs* according to the pseudo-condition N. If the indicated condition is satisfied or if N is 15 (*always*), adds the 8-bit signed displacement to the contents of the Program Counter to determine the destination address; and places this address in the Program counter. Otherwise, executes the next sequential instruction.

Addressing: Source, mode 5;
Destination, abbreviated mode 2

Bytes: 3

Cycles: 11, if condition is met; otherwise, 10

Conditions: Unchanged

Examples: if (bit(0,*b1)) goto dest;
if (!bit(0,*b2)) goto dest;

Branch on Memory Bit (T/F) and Increment

True -

*if (bit(N,*bs++)) goto symbol;*

0111	1011
ssss	nnnn
offset	

Hex 7B
sN
offset

False -

*if (!bit(N,*bs++)) goto symbol;*

0111	0011
ssss	nnnn
offset	

Hex 73
sN
offset

Operation

If [!]bit(N,source) then PC ← destination address

Tests the 8-bit contents of memory location _*bs_ according to the pseudo-condition N. If the indicated condition is satisfied or if N is 15 (_always_), adds the 8-bit signed displacement to the contents of the Program Counter to determine the destination address; places this address in the Program Counter; and increments the contents of the designated B Register. Otherwise, executes the next sequential instruction.

Addressing:	Source, mode 7; Destination, abbreviated mode 2
Bytes:	3
Cycles:	14
Conditions:	Unchanged
Examples:	if (bit(7,*b2++)) goto dest; if (!bit(7,*b3++)) goto dest;

3.5 Special Register Instructions

These instructions deal with special processor registers — the Condition Register (CR), the Register Pointer (RP), the Stack Pointer (SP) — and tend to effect significant changes in the state of the program. Instructions in this group that address memory use the same address modes as the dyadic instructions.

3.5.1 Load Instructions

These instructions permit 16-bit addresses to be loaded into a B Register, the RP, or the SP. None of these instructions affect the condition bits. Each description illustrates the only valid form of the instruction.

Load Address

_bd = &*(bs + n);_

0110	1111
dddd	ssss
offset	

Hex 6F
ds
offset

Operation

destination ← address of source

Loads the address of the source into the destination B Register. Computes the operand addresses in the same manner as for other dyadic instructions of the same mode. When the s field contains 15 and the offset is 0, this instruction loads the 16-bit contents of the Stack Pointer into a B Register.

Addressing: Dyadic mode 6

Bytes: 3

Cycles: 13

Conditions: Unchanged

Examples: b5 =&*(b7 +2);
b7 =&*(sp +0);

Load Register Address

bd = &bs;

0110	1101
dddd	ssss

Hex 6D
ds

Operation

destination ← address of source

Loads the address of the source into the destination B Register. Computes the operand addresses in the same manner as for other dyadic instructions of the same mode. In addition to loading the absolute address of any normally addressable memory location, this instruction makes it possible to load the 16-bit contents of the Register Pointer (when the s field contains 0) or of the Program Counter (when the s field contains 15) into a B Register.

Addressing: Dyadic mode 0

Bytes: 2

Cycles: 9

Conditions: Unchanged

Examples: b2 =&b12;
b10 =&*rp;
b0 =&*pc;

Load Register Pointer

rp = bs;

0100	1101
0000	ssss

Hex 4D
0s

Operation

RP ← source ANDed with X(FFF8)

Loads bits <15,3> of the Register Pointer with the corresponding bits of the source and sets bits <2,0> of the Register Pointer to zero. The source is not altered.

Addressing: Source, 16-bit dyadic mode 0

Bytes: 2 to 4

Cycles: 8

Conditions: Unchanged

Examples: rp =b6;
rp =32678;

Load Stack Pointer

sp = bs;

0000	1101
0000	ssss

Hex 0D
0s

Operation

SP ← source

Loads the Stack Pointer with the 16-bit contents of the source. The source is not changed.

Addressing: Source, 16-bit dyadic mode 0

Bytes: 2 to 4

Cycles: 8

Conditions: Unchanged

Examples: sp =b2;
sp =1375;

3.5.2 Condition Register Instructions

These instructions modify the CR under direct program control. Each description illustrates the only valid form of the instruction.

Clear Conditions

clear(n);

or

clear(zero,carry, ...);

0000	0011
data	

Hex 03
data

Operation

CR ← CR ANDed with complement of mask

Clears selected condition bits in the Condition Register using the 8-bit binary number *n* or any selection of the first eight condition names (see **Conditions**) as a mask (data). Bits that are 1 in the mask are forced to 0 (false) in the Condition Register; other condition bits are unchanged. When the condition names are used in the assembly language coding, the assembler determines the value of the mask (data).

Bytes: 2

Cycles: 5

Conditions: Unchanged except as indicated by mask

Examples: clear(flag);
clear(6);
clear(zero,carry);

Set Conditions

set (n);

or

set(zero,carry, ...);

0000	0001
data	

Hex 01
data

Operation

CR ← CR ORed with mask

Sets the selected condition bits true in the Condition Register using the 8-bit binary number *n* or a selection of the first eight condition names as a mask (data). Bits that are 1 in the mask are forced to 1 (true) in the Condition Register; other condition bits are unchanged. When the condition names are used in the assembly language coding, the assembler determines the value of the mask (data).

Bytes: 2

Cycles: 5

Conditions: Unchanged, except as indicated by mask

Examples: set(ovfl);
set(7);
set(neg,carry);

3.5.3 Register Pointer Instructions

These instructions modify the RP under program control. Because the RP contains the starting address in memory of the general registers, modifying the RP effectively shifts these registers up or down in memory. The contents of the general registers are unchanged but the registers are addressed differently following execution of any instruction in this group. None of these instructions affect the condition bits. Each description illustrates the only valid form of the instruction.

Bump 4

rp = rp + 8;

0100	0010

Hex 42

Operation

RP ← RP +8

Increases the contents of the Register Pointer by 8. This moves the general registers up in memory by four positions to provide four new registers addressable as A12/B12 through A15/B15. The original registers A0/B0 through A3/B3 no longer are part of the register set, and the original registers A4/B4 through A15/B15 now are addressable as registers A0/B0 through A11/B11.

Bytes: 1

Cycles: 6

Conditions: Unchanged

Example: rp =rp +8;

Bump 8

rp = rp +16;

0100	0011

Hex 43

Operation

RP ← RP +16

Increases the contents of the Register Pointer by 16. This moves the general registers up in memory by eight positions to provide eight new registers addressable as A8/B8 through A15/B15. The original registers A0/B0 through A7/B7 no longer are part of the register set, and the original registers A8/B8 through A15/B15 now are addressable as A0/B0 through A7/B7.

Bytes: 1

Cycles: 6

Conditions: Unchanged

Example: rp =rp +16;

Debump 4

rp = rp − 8;

0100	1010

Hex 4A

Operation

RP ← RP −8

Decreases the contents of the Register Pointer by 8. This moves the general registers down in memory by four positions to provide four new registers addressable as A0/B0 through A3/B3. The original registers A12/B12 through A15/B15 no longer are part of the register set, and the original registers A0/B0 through A11/B11 are addressable as A4/B4 through A15/B15.

Bytes:	1
Cycles:	6
Conditions:	Unchanged
Example:	rp =rp −8;

Debump 8

rp = rp − 16;

0100	1011

Hex 4B

Operation

RP ← RP −16

Decreases the contents of the Register Pointer by 16. This moves the general registers down in memory by eight positions to provide eight new registers addressable as A0/B0 through A7/B7. The original registers A8/B8 through A15/B15 no longer are part of the register set, and the original registers A0/B0 through A7/B7 now are addressable as A8/B8 through A15/B15.

Bytes:	1
Cycles:	6
Conditions:	Unchanged
Example:	rp =rp −16;

3.5.4 Stack Instructions

These instructions permit data to be stored in and retrieved from the stack under program control. None of the instructions in this group affect the condition bits with the exception of restoring the CR. In this case, the CR takes on the condition of the restored byte. Each description illustrates the only valid form of the instruction.

Pop A

ad = pop();

0000	0100
dddd	0000

Hex 04
d0

Operation

destination ← *SP;
SP ← SP + 1

Pops the byte from the top of the stack and copies it into the destination A Register. If *ad* is 15, the destination is register A15. Increments the Stack Pointer by 1.

Addressing: Monadic mode 0

Bytes: 2

Cycles: 7

Conditions: Unchanged

Example: a0 = pop();

Pop B

bd = pop();

0100	0100
dddd	0000

Hex 44
d0

Operation

destination ← *SP;
SP ← SP + 2

Pops two bytes from the stack. Copies the byte from the top of the stack into the low-order byte of the destination B Register and the byte from the next higher stack address into the high-order byte. If *bd* is 15, the destination is register B15. Increments the Stack Pointer by 2.

Addressing: Monadic mode 0

Bytes: 2

Cycles: 10

Conditions: Unchanged

Example: b14 = pop();

Pop Condition Register

cr = pop();

0000	0101

Hex 05

Operation

CR ← *SP;
SP ← SP +1

Pops the byte from the top of the stack and copies it into the Condition Register. Increments the Stack Pointer by 1.

Bytes:	1
Cycles:	7
Conditions:	Determined by the byte popped
Example:	cr =pop();

Pop Register Pointer

rp = pop();

0100	0101

Hex 45

Operation

RP ← *SP;
SP ← SP +2

Pops two bytes from the stack. Copies the byte from the top of the stack into the low-order byte of the Register Pointer and the byte from the next higher stack address into the high-order byte. Increments the Stack Pointer by 2.

Bytes:	1
Cycles:	9
Conditions:	Unchanged
Example:	rp =pop();

Push A

push(as);

0000	0100
0000	ssss

Hex 06
0s

Operation

```
SP ← SP − 1;
*SP ← source
```

Decrements the Stack Pointer by 1. Stores the 8-bit contents of the source A Register on the top of the stack. If *as* is 15, the contents of register A15 are pushed onto the stack. The contents of *as* are not changed.

Addressing: Source addressed as dyadic mode 0

Bytes: 2

Cycles: 6

Conditions: Unchanged

Example: push(a4);

Push B

push(bs);

0100	0110
0000	ssss

Hex 46
0s

Operation

```
SP ← SP − 2;
*SP ← source
```

Decrements the Stack Pointer by 2. Stores the 16-bit contents of the source B Register on the stack. Pushes the high-order byte of register *bs* first. If *bs* is 15, the contents of register B15 are pushed. The contents of *bs* are unchanged.

Addressing: Source addressed as dyadic mode 0

Bytes: 2

Cycles: 10

Conditions: Unchanged

Example: push(b7);

Push Condition Register

push(cr);

0000	0111

Hex 07

Operation

SP ← SP − 1;
*SP ← CR

Decrements the Stack Pointer by 1. Stores the 8-bit contents of the Condition Register on the top of the stack. The contents of the Condition Register are not changed.

Bytes: 1

Cycles: 6

Conditions: Unchanged

Example: push(cr);

Push Register Pointer

push(rp);

0100	0111

Hex 47

Operation

SP ← SP − 2;
*SP ← RP

Decrements the Stack Pointer by 2. Stores the 16-bit contents of the Register Pointer on the stack. Pushes the high-order byte first. The contents of the Register Pointer are not changed.

Bytes: 1

Cycles: 8

Conditions: Unchanged

Example: push(rp);

3.5.5 Add 8-bits Instructions

These instructions permit altering a 16-bit register (B Register or the SP) with 8-bit data from an A Register or contained in the instruction. Each description illustrates the valid forms of the instruction.

Arithmetic Add 8 Bits

bd = bd + as;

or

sp = sp + as;

0111	1101
dddd	ssss

Hex 7D
ds

Operation

Bd ← Bd + As

or

SP ← SP + As

Treats the source and destination operands as signed two's-complement numbers. To add the 8-bit source number to the 16-bit destination, the processor expands the 8-bit number to 16 bits by filling the high-order byte with the sign bit of the low-order byte, thus retaining the signed value of this number in the 16-bit form. The source is not changed. For the case where the d field is 15, the destination operand is the Stack Pointer.

Addressing:	Dyadic mode 0; except when d is 15, destination is the SP.
Bytes:	2 to 3
Cycles:	14
Conditions:	Unchanged
Examples:	b6 = b6 + a4; sp = sp + a0;

Logical Add 8 Bits

bd = bd + logical(as);

or

sp = sp + logical(as);

0111	0101
dddd	ssss

Hex 75
ds

Operation

$$Bd \leftarrow Bd + As$$

or

$$SP \leftarrow SP + As$$

Treats the source and destination operands as unsigned numbers. To add the 8-bit source number to the 16-bit destination, the processor expands the 8-bit number to 16 bits by filling the high-order byte with zeros, thus retaining the unsigned value of this number in the 16-bit form. The source is not changed. For the case where the d field is 15, the destination operand is the Stack Pointer.

Addressing: Dyadic mode 0; except when d is 15, destination is the SP.

Bytes: 2 to 3

Cycles: 14

Conditions: Unchanged

Examples: b6 = b6 + logical(a4);
sp = sp + logical(a14);

3.6 Miscellaneous Instructions

These instructions do not fit into any of the other categories. They consist of monadic or dyadic instructions that do not use the standard instruction format or instructions that affect the general processing.

The operation to be performed is designated by an *operator name* rather than an operator symbol; refer to **Tokens** in Chapter 6. Each description illustrates the valid form of the instruction.

Count Ones

ad = bitsum(as);

0000	1110
dddd	ssss

Hex 0E
ds

Operation

destination ← number of 1 bits in source

Replaces the contents of the destination operand with a binary number that is equal to the number of 1 bits in the source. The source is not altered (unless the s and d fields specify the same register).

Addressing:	8-bit dyadic mode 0
Bytes:	2
Cycles:	8
Conditions:	neg, zero, ones, odd
Example:	a5 =bitsum(a3);

Find Leftmost One

ad = flo(as);

0000	1100
dddd	ssss

Hex 0C
ds

Operation

destination ← index of leftmost 1 bit in source

Replaces the contents of the destination operand with a binary number that indicates the bit position of the leftmost 1 bit in the source. The least-significant bit (bit 0) in the source corresponds to a destination value of 1; the most-significant bit (bit 7) corresponds to a destination value of 8. If the source contains all zeros, the destination is set to 0. The source is not altered (unless the s and d fields specify the same register).

Addressing:	8-bit dyadic mode 0
Bytes:	2
Cycles:	8
Conditions:	neg, zero, ones, odd
Example:	a5 =flo(a7);

Find and Clear Leftmost One

ad = floc(as);

0100	1100
dddd	ssss

Hex 4C
ds

Operation

```
destination ← index of leftmost 1 bit in source;
source<leftmost 1 bit> ← 0
```

Replaces the contents of the destination operand with a binary number that indicates the bit position of the leftmost 1 bit in the source. The least-significant bit (bit 0) in the source corresponds to a destination value of 1; the most-significant bit (bit 7) corresponds to a destination value of 8. If the source contains all zeros, the destination is set to 0. The bit that is found is set to 0 in the source.

Addressing:	8-bit dyadic mode 0
Bytes:	2
Cycles:	11
Conditions:	neg, zero, ones, odd
Example:	a5 =floc(a7);

Sign Extend

extend(bd);

0110	0010
dddd	0000

Hex 62
d0

Operation

```
If destination<7> is 0,
  destination <15.8> ← X(00);
else destination <15.8> ← X(FF)
```

Fills the high-order byte of the destination B Register with the sign bit (most-significant bit) of the low-order byte.

Addressing:	Monadic mode 0

Bytes: 2

Cycles: 6

Conditions: Unchanged

Example: extend(b6);

Swap Bytes

swap(bd);

0110	1010
dddd	0000

Hex 6A
d0

Operation

$$\text{destination}<7,0> \rightleftarrows \text{destination}<15,8>$$

Exchanges the high- and low-order bytes of the destination B Register.

Addressing: Monadic mode 0

Bytes: 2

Cycles: 9

Conditions: Unchanged

Example: swap(b13);

Halt

halt();

0111	1000

Hex 78

Operation

$$\text{CR}<6> \leftarrow 1$$

Halts program execution and sets the enable condition bit (CR<6>). The address and data buses and the read and write drivers are in a high-impedance mode. This state continues until the processor receives an interrupt request or reset signal.

Bytes: 1

Cycles: 4 (minimum)

Conditions: enable set, others unchanged

Example: halt();

No Op

nop();

0111	1111

Hex 7F

Operation

none

Provides a 4-clock-cycle delay during which no operation is performed. Execution then continues with the next instruction.

Bytes: 1

Cycles: 4

Conditions: Unchanged

Example: nop();

Chapter 4

SPECIFICATIONS

4. SPECIFICATIONS

This chapter contains the operating characteristics, input and output waveforms, and package (external) dimensions for the MAC-8 microprocessor.

4.1 Operating Characteristics

The MAC-8 is available in two versions. The major difference between the versions is the speed of the device. This is determined by the chip voltage.

A standard MAC-8 operates at 2 MHz with a chip voltage of +12V±5 percent. It can accommodate slower storage and system elements if wait states are implemented. Wait states affect only the READ and WRITE access times not the MAC-8's internal processing speed. Figures 2-6 and 2-7 illustrate the differences between READ or WRITE operations for fast and slow memory access, respectively.

If required for the system design, slower versions of the MAC-8 are available. The chip voltage, which can vary from +8.5V to +12.6V, determines the speed of the processor. The slower MAC-8 easily handles faster memory and peripherals. If slower memory and system elements must be accommodated, wait states also must be implemented.

Specifications for both versions of the MAC-8 are provided here. The waveforms in the next subsection illustrate the timing parameters included in the specifications. Both versions have the same external dimensions.

4.1.1 Standard MAC-8 (2-MHz Operation)

RECOMMENDED OPERATING CONDITIONS					
Symbol	**Parameter**	**Minimum**	**Nominal**	**Maximum**	**Unit**
V_{CC}	Supply voltage, TTL circuits, pin 5	+4.75	+5.0	+5.25	V
V_{DD}	Supply voltage, chip, pin 26	+11.4	+12.0	+12.6	V
V_{SS}	Supply voltage, ground, pin 13		0		V
C_{LOAD}	Output capacitance		25		pF
T_A	Operating temperature	-40		+85	°C

TIMING CHARACTERISTICS				
Symbol	**Parameter**	**Minimum**	**Maximum**	**Unit**
T_{in}	Pulse width, CLK IN low	243		ns
T'_{in}	Pulse width, CLK IN high	243		ns
T_{out}	Pulse width, CLK OUT low	$T_{in}+10$	$T_{in}+110$ or $T_{in}+T_{in}-300$ (Note 1)	ns
T'_{out}	Pulse width, CLK OUT high	$T'_{in}-110$ or 300 (Note 2)	$T'_{in}-10$ or 1500 (Note 2)	ns
T_{ac}	Transition time, CLK OUT low to ADDRESS	0	-	ns
T'_{ac}	Time, CLK OUT (low) to ADDRESS VALID	0	60	ns
T_{crd}	Delay to change of $\overline{\text{READ}}$ or STATUS	-	85	ns
T_{access}	Access time allowed for off-chip memory (READ)	$T_{in}+T'_{in}-150$	-	ns
T_{ds}	DATA (READ) setup time	-	160	ns
T_{dh}	DATA (READ) hold time	0		ns
T_{cw}	Delay after memory $\overline{\text{WRITE}}$ starts	40 (Note 3)	200	ns
T_{wc}	Delay after memory $\overline{\text{WRITE}}$ ends	5	125	ns
T_{wd}	Time available for off-chip memory WRITE	$T_{in}+T'_{in}-290$	-	ns

TIMING CHARACTERISTICS

Symbol	Parameter	Minimum	Maximum	Unit
T_{wdh}	DATA VALID (WRITE) to CLK OUT high	0	-	ns
T_{wds}	DATA (WRITE) out delay	-	235	ns
T_{wdw}	DATA (WRITE) hold time	0	-	ns
T_{wp}	Memory WRITE pulse width	225	-	ns
T_{mracc}	Time allowed for DATA READY in	$T_{in} - 100$	-	ns
T_{ryh}	DATA READY hold time	0	-	ns
T_{is}	INTERRUPT or RESET set-up time	-	80	ns
T_{ih}	INTERRUPT or RESET hold time	150	-	ns
T_{dmas}	$\overline{\text{DMA REG}}$ (request) setup time	-	200	ns
T_{dmah}	$\overline{\text{DMA REQ}}$ (request) hold time	0	-	ns
T_{dmak}	$\overline{\text{DMA ACK}}$ (acknowledge) delay (on)	-110	100	ns
T_{dmax}	$\overline{\text{DMA ACK}}$ (acknowledge) delay (off)	-110	100	ns
T_{dmaa}	Start of ADDRESS change to DMA ADDRESS float	0	-	ns
T_{dma}	ADDRESS delay in DMA mode (off)	160	-	ns
T'_{dma}	ADDRESS delay in DMA mode (on)	-	160	ns
T_r,T_f	Clock rise and fall times	-	1	ns/pF
T_{c1}	Delay time, CLK IN low to CLK OUT low	-	230	ns
T_{c2}	Delay time, CLK IN high to CLK OUT high	-	290	ns

NOTES:

1. Whichever is greater.
2. Whichever is less.
3. If $T_A<0^\circ C$, T_{cw} (minimum)$=50$ ns.

dc CHARACTERISTICS					
Symbol	**Parameter**	**Conditions**	**Minimum**	**Maximum**	**Unit**
I_{OL}	Output sink current	V_{OUT}=+0.4V	1.6		mA
		V_{OUT}=+0.5V	2.0		mA
I_{OH}	Output source current	V_{OUT}=+3.2V	160		μA
I_L	Leakage current, at input or at high impedance output	V_{IN} or V_{OUT}=+5.25V		400	nA
		V_{IN} or V_{OUT}=0V	-400		nA
I_{DD}	V_{DD} supply current	F=0 Hz, ALU on		10	mA
		F=0 Hz, ALU off		5	mA
		F=2 MHz, V_{DD}=+12.6V		45	mA
I_{CC}	V_{CC} supply current	No dc load on any output		2	mA
V_{INH}	Input, one (high) level		+1.8		V
V_{INL}	Input, zero (low) level			+0.8	V

4.1.2 Slower MAC-8

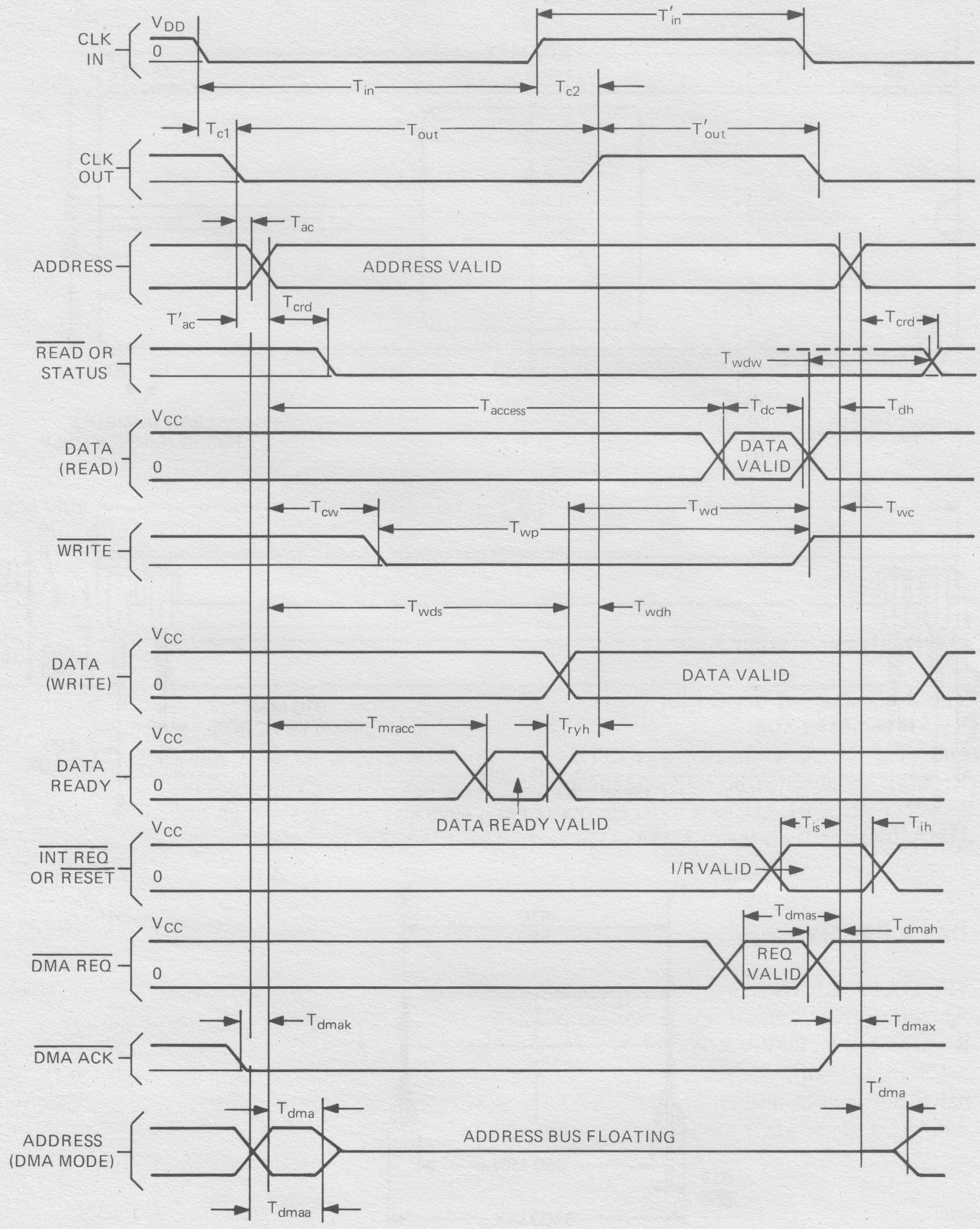
RECOMMENDED OPERATING CONDITIONS					
Symbol	**Parameter**	**Minimum**	**Nominal**	**Maximum**	**Unit**
V_{CC}	Supply voltage, TTL circuits, pin 5	+4.75	+5.0	+5.25	V
V_{DD}	Supply voltage, chip, pin 26	+8.5		+12.6	V
V_{SS}	Supply voltage, ground, pin 13		0		V
C_{LOAD}	Output capacitance		25		pF
T_A	Operating temperature	-40		+85	°C

TIMING CHARACTERISTICS				
Symbol	**Parameter**	**Minimum**	**Maximum**	**Unit**
T_{in}	Pulse width, CLK IN low	480		ns
T'_{in}	Pulse width, CLK IN high	480		ns
T_{out}	Pulse width, CLK OUT low	$T_{in}+20$	$T_{in}+220$	ns
T'_{out}	Pulse width, CLK OUT high	$T'_{in}-220$	$T'_{in}-20$	ns
T_{ac}	Transition time, CLK OUT low to ADDRESS	0	70	ns
T'_{ac}	Time, CLK OUT low to ADDRESS VALID	0	120	ns
T_{crd}	Delay time to change of $\overline{\text{READ}}$ or STATUS		170	ns
T_{access}	Access time allowed for off-chip memory READ	$T_{in}+T'_{in}-300$		ns
T_{ds}	DATA (READ) setup time		320	ns
T_{dh}	DATA (READ) hold time	0		ns
T_{cw}	Delay after memory WRITE starts		400	ns
T_{wc}	Delay after memory WRITE ends	0	200	ns
T_{wd}	Time available for off-chip memory WRITE	$T_{in}+T'_{in}-580$		ns
T_{wdh}	DATA VALID (WRITE) to CLK OUT high	0		ns
T_{wds}	DATA (WRITE) out delay		470	ns
T_{wda}	DATA (WRITE) hold time	0		ns
T_{mracc}	Time allowed for DATA READY in	$T_{in}-200$		ns
T_{ryh}	DATA READY hold time	0		ns

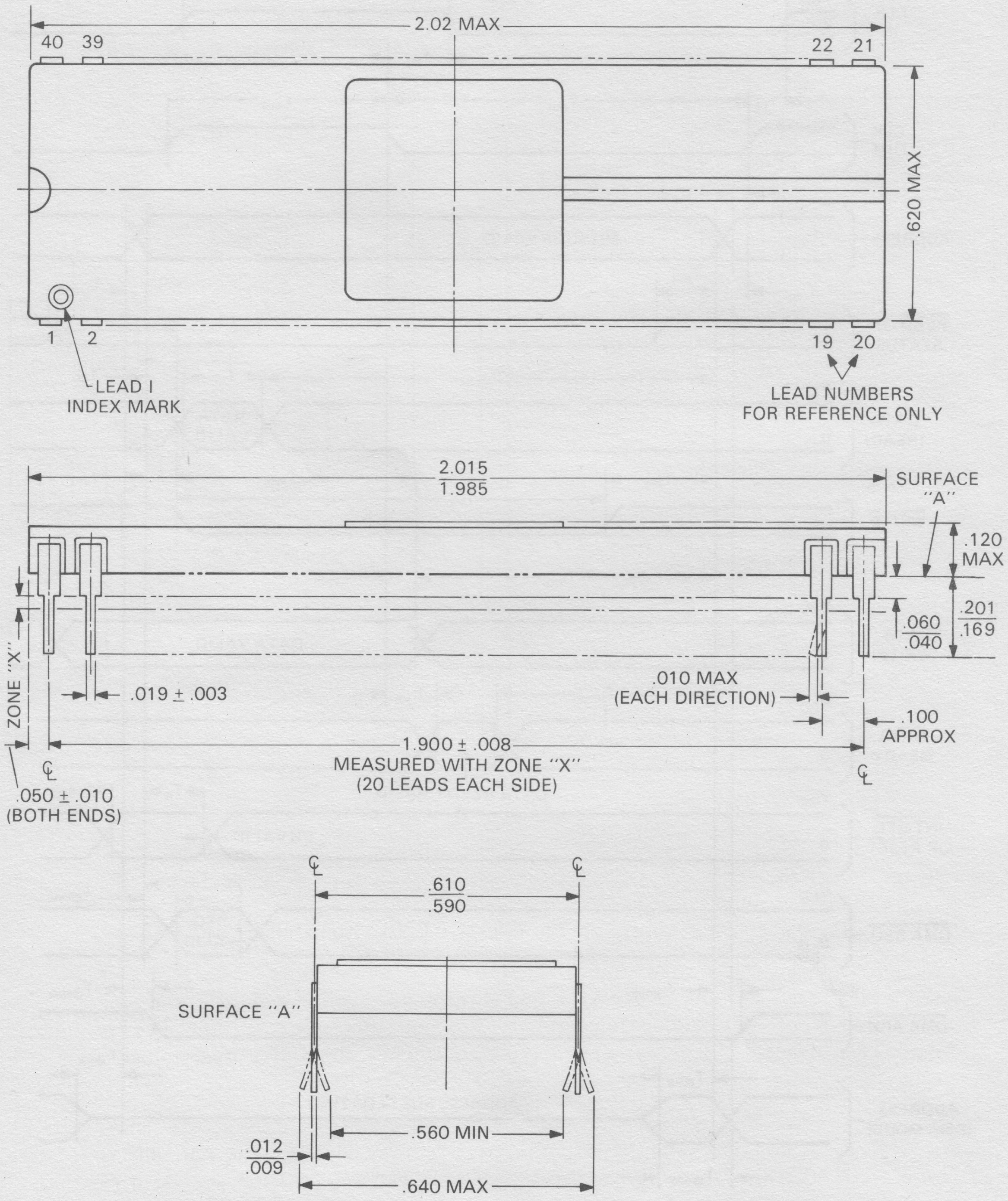
TIMING CHARACTERISTICS				
Symbol	**Parameter**	**Minimum**	**Maximum**	**Unit**
T_{is}	INTERRUPT or RESET set-up time		160	ns
T_{ih}	INTERRUPT or RESET	300		ns
T_{dmas}	$\overline{\text{DMA REQ}}$ (request) setup time		400	ns
T_{dmah}	$\overline{\text{DMA REQ}}$ (request) hold time	0		ns
T_{dmak}	$\overline{\text{DMA ACK}}$ (acknowledge) delay (on)	-220	200	ns
T_{dmax}	$\overline{\text{DMA ACK}}$ (acknowledge) delay (off)	-220	200	ns
T_{dmaa}	Start of ADDRESS change to DMA ADDRESS float	0		ns
T_{dma}	ADDRESS delay in DMA mode (off)	280	60	ns
T'_{dma}	ADDRESS delay in DMA mode (on)		320	ns
T_r,T_f	Clock rise and fall times		2.5	ns/pF

dc CHARACTERISTICS					
Symbol	**Parameter**	**Conditions**	**Minimum**	**Maximum**	**Unit**
I_{OL}	Output sink current	$V_{OUT}=+0.4V$	1		mA
I_{OH}	Output source current	$V_{OUT}=+2.4V$	160		μA
I_L	Leakage current, at input or at high impedance output	V_{IN} or $V_{OUT}=+5.25V$		200	nA
		V_{IN} or $V_{OUT}=0V$		200	nA
I_{DD}	V_{DD} supply current	ALU on		10	mA
		ALU off		5	mA
I_{CC}	V_{CC} supply current	No dc load on any output		2	mA
V_{INH}	Input one (high) level		+1.8		V
V_{INL}	Input zero (low) level			+0.8	V
V_{TH}	Power-on reset level		+1.5	+4.0	V

4.2 Input and Output Waveform


CLK IN
V_{DD}
0
T'_{in}
T_{in}
T_{c2}
T_{c1}
T_{out}
T'_{out}
CLK OUT
T_{ac}
ADDRESS
ADDRESS VALID
T'_{ac}
T_{crd}
T_{crd}
READ OR STATUS
T_{wdw}
T_{access}
T_{dc}
T_{dh}
V_{CC}
DATA (READ)
0
DATA VALID
T_{cw}
T_{wd}
T_{wc}
T_{wp}
WRITE
T_{wds}
T_{wdh}
V_{CC}
DATA (WRITE)
0
DATA VALID
T_{mracc}
T_{ryh}
V_{CC}
DATA READY
0
DATA READY VALID
T_{is}
T_{ih}
V_{CC}
INT REQ OR RESET
0
I/R VALID
T_{dmas}
T_{dmah}
V_{CC}
DMA REQ
0
REQ VALID
T_{dmak}
T_{dmax}
DMA ACK
T'_{dma}
T_{dma}
ADDRESS (DMA MODE)
ADDRESS BUS FLOATING
T_{dmaa}

4.3 Package Dimensions


2.02 MAX
40
39
22
21
.620 MAX
1
2
19
20
LEAD I INDEX MARK
LEAD NUMBERS FOR REFERENCE ONLY
2.015
1.985
SURFACE "A"
.120 MAX
.201
.169
.060
.040
ZONE "X"
.019 ± .003
.010 MAX (EACH DIRECTION)
.100 APPROX
1.900 ± .008
MEASURED WITH ZONE "X" (20 LEADS EACH SIDE)
℄
.050 ± .010 (BOTH ENDS)
.610
.590
SURFACE "A"
.560 MIN
.012
.009
.640 MAX

Preliminary

Chapter 5

SYSTEM OPERATION

5. SYSTEM OPERATION

Some aspects of system design (e.g., interfacing, peripheral devices, and system integration) are affected by the operating characteristics and requirements of a standard MAC-8 microprocessor.

5.1 Interfacing

The operating characteristics and pin functions of the MAC-8 can influence the system design. In particular the system designer should be concerned with the timing requirements of the system elements (e.g., memory, input/output) and limitations that the operating characteristics[1] of the MAC-8 microprocessor impose on the system.

5.1.1 Access Timing

The basic READ or WRITE access timing are the key factors for the MAC-8 microprocessor. Direct memory access (DMA) timing must also be considered. All parameters identified in this section are described and illustrated in Chapter 4.

5.1.1.1 READ Timing

The chip selection method determines the time frame for valid data to appear on the data bus during a READ access:

- *ADDRESS only* — Of primary concern with this method is the T_{access} parameter, which is measured from ADDRESS VALID to the beginning of the Data Latch Strobe. The nominal T_{access} time is 350 ns. Assuming zero wait states, this means a 350-ns time frame exists for valid data to appear on the data bus once the MAC-8 establishes an address.
- *Combination of ADDRESS and* $\overline{\text{READ}}$ *signals* — With this method, a nominal 265-ns time frame exists for valid data to appear on the data bus. If the user's design cannot yield system elements with access times that are adequate for this time frame, wait states must be implemented.

The Data Latch Strobe (T_{ds}) period can be calculated by

$$T_{ds} = T_{cycle} - T_{access} - (T_r \text{ and } T_f)_{address}$$

$$125 \text{ ns} = 500 \text{ ns} - 350 \text{ ns} - 25 \text{ ns}.$$

Toward the end of this 125-ns time frame, the contents of the data bus will be latched internal to the MAC-8 processor. A zero DATA HOLD (T_{dh}) time exists beyond this point.

---

[1] Refer to Chapter 4 for the MAC-8 specifications and waveform.

When the READ cycle terminates, the $\overline{\text{READ}}$ signal level returns to the non-active state (i.e., high logic level) a maximum of 85 ns beyond the ADDRESS stable of the next cycle; refer to the T_{crd} specification in Chapter 4.

The $\overline{\text{READ}}$ signal functions as a level *not* a strobe. This can be readily understood by considering successive reads. Here, no reclocking of the $\overline{\text{READ}}$ signal occurs; its level is maintained in the presence of address bus switching.

5.1.1.2 WRITE Timing

The major timing considerations for the WRITE operation are:

T_{wd}	Time available to WRITE to off-chip memory
T_{cw}	DATA VALID point
T_{wc}	Delay after WRITE terminates
T_{wds}	Delay from memory WRITE end to ADDRESS change
T_{wda}	DATA HOLD time.

When ADDRESS is the chip selection method, a 235-ns maximum time frame exists before data is available on the data bus. When both ADDRESS and the $\overline{\text{WRITE}}$ strobe are used for chip select, the delay to the start of a memory WRITE (i.e., the T_{cw} parameter) maximally reduces this time to 75 ns. The T_{cw} specification is 0 minimum, 160 ns maximum.

At the DATA VALID point, the $\overline{\text{WRITE}}$ strobe remains active (low logic level) for 196 ns minimum, which is the time available to WRITE off-chip memory (T_{wd} specification). In addition, the low to high transition of CLK OUT (the T_{wdh} parameter) acts as a reference for DATA (WRITE) stable on the data bus. The DATA HOLD time (T_{wda} specification) after termination of the WRITE cycle is a minimum of 0 ns.

Upon termination of the WRITE cycle, a delay of 0 ns minimum to 100 ns maximum exists before ADDRESS change. For successive writes, a minimum of 5-ns separation can exist between $\overline{\text{WRITE}}$ strobes.

5.1.1.3 DMA Timing

When using the DMA function of the MAC-8 microprocessor, a potential problem area exists as a result of the T'_{dma} specification, i.e., the address delay in the DMA mode. This 160-ns maximum delay cuts into the access time of the opcode fetch cycle when *coming out* of the DMA mode. A widely used remedy is to insert a wait-state cycle, which defers the actual opcode access by one cycle.

5.1.2 Wait State Generation

If storage and system elements cannot use the *raw* MAC-8 access times, wait-state cycles must be implemented. The DATA READY input signal on pin 35 of the MAC-8 provides this function.

The MAC-8 strobes the DATA READY line at mid-cycle. An active (low logic level) signal on the DATA READY pin causes the MAC-8 to suspend further processing operations and maintain the logical conditions that exist on the address and data buses and on the $\overline{\text{READ}}$ and $\overline{\text{WRITE}}$ lines. The CLK OUT signal (pin 32) remains normal during wait cycles. This suspended state continues until the DATA READY line strobes *high* at mid-cycle at which time normal processing resumes.

The Western Electric (WE) 146D peripheral timer chip, which is described in this chapter (see **Peripherals**), provides a single wait-state cycle for the MAC-8.

5.1.2.1 DATA READY Strobe

The minimum time allowed for DATA READY setup (T_{mracc}) is 143 ns, measured from the ADDRESS stable point (T_{ac} specification).

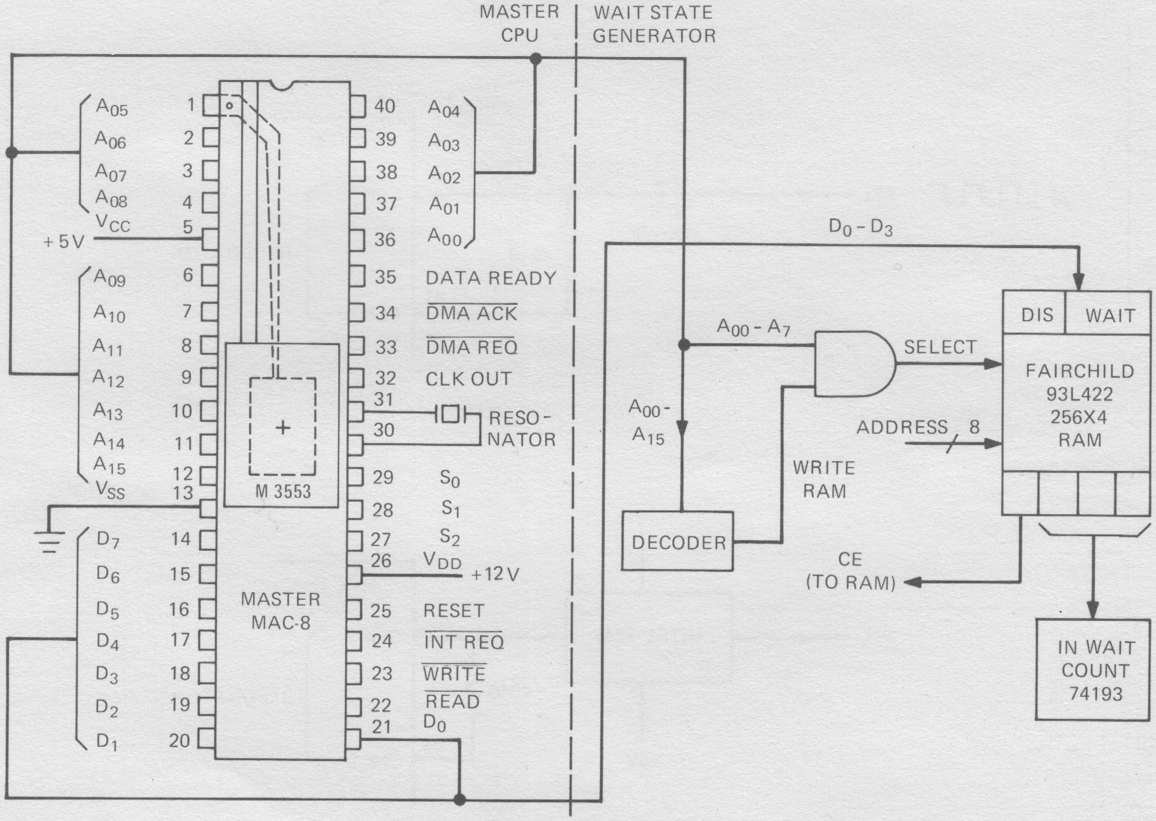

MASTER CPU
WAIT STATE GENERATOR
A05
A06
A07
A08
VCC
+5V
A09
A10
A11
A12
A13
A14
A15
VSS
D7
D6
D5
D4
D3
D2
D1
A04
A03
A02
A01
A00
DATA READY
DMA ACK
DMA REQ
CLK OUT
RESO-NATOR
S0
S1
S2
VDD
+12V
RESET
INT REQ
WRITE
READ
D0
M 3553
MASTER MAC-8
D0 - D3
A00 - A7
SELECT
A00 - A15
ADDRESS 8
WRITE RAM
DECODER
CE (TO RAM)
DIS
WAIT
FAIRCHILD 93L422 256X4 RAM
IN WAIT COUNT 74193

Figure 5-1. Programmable Wait-State Generator

5.1.2.2 DATA READY Control

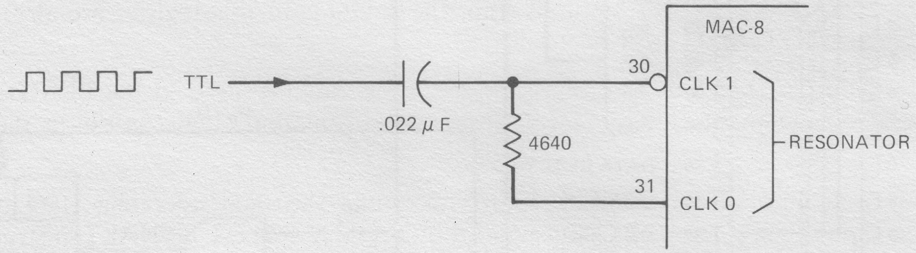
Two methods are available for controlling the DATA READY signal using wait states:

1. *Synchronous* — preallocate an integral number of wait cycles for each access or block of access. The DATA READY line is maintained in the logic zero state for the number of wait cycles requested. Operation proceeds with the DATA READY line in the logic one state. Figure 5-1 is an example of a programmable wait-state generator.

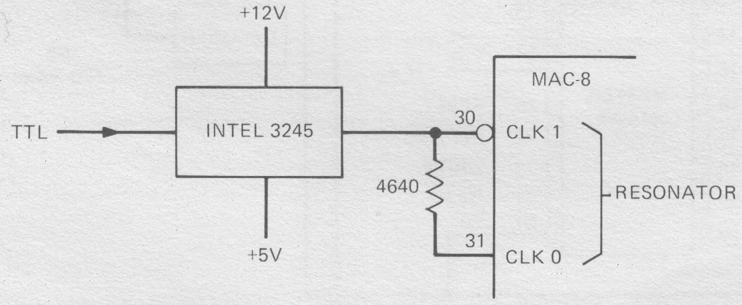
2. *Asynchronous* — generate a varied number of wait cycles for each access determined by the system element's response. The DATA READY line normally is zero and under the control of the accessed system element. At the discretion of this system element, the DATA READY line will be sent high thus terminating the access. For each new access, the DATA READY line is zero and the process is repeated.

5.1.3 CLK IN Control

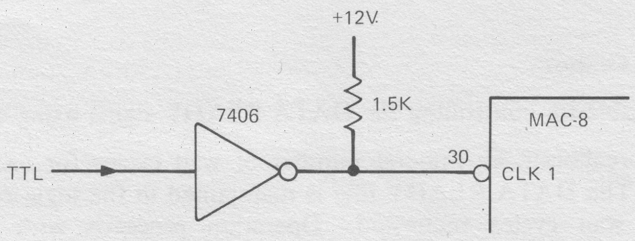
Because the MAC-8 uses silicon-gate CMOS technology, it requires an external timing reference (clock) to refresh and provide timing control signals to the internal storage elements and logic circuitry. A single-phase clock must be connected in series to the MAC-8 resonator pins 30 and 31. This CLK IN waveform should comply with the electrical specifications provided in Chapter 4. Figure 5-2 illustrates several methods for generating the clock control.


TTL
.022 μ F
4640
MAC-8
30
CLK 1
31
CLK 0
RESONATOR

A. Internal Clock Generator


+12V
TTL
INTEL 3245
+5V
4640
MAC-8
30
CLK 1
31
CLK 0
RESONATOR

B. Alternate Internal Clock Generator


+12V
1.5K
7406
TTL
MAC-8
30
CLK 1

C. External Clock Input

Figure 5-2. MAC-8 Clock Generator

5.1.4 MAC-8 Status

The status information supplied on pins 27, 28, and 29 provides the operating status pending for the next MAC-8 clock cycle (also refer to **CPU Status** in Chapter 2).

One function of the status pins allows the MAC-8 on-chip registers — i.e., Condition Register (CR), Instruction Register (IR), Program Counter (PC), Register Pointer (RP), and Stack Pointer (SP) — to be monitored for program diagnostics and tracing. During certain instruction cycles when executing MAC-8 code, the contents of the on-chip registers appear on the address or data bus. The status information (except for the CR status indication) is available a half cycle before the contents actually appear on the bus, which allows monitoring hardware to be set up and triggered. When the CR status is true, the contents of this register are already on the data bus; refer to **CPU Status** in Chapter 2.

For example, Figure 5-3 illustrates the register-monitoring circuitry used in the PLAID (Program Logic Aid) development system. Decoded status information allows access to the CR, PC, RP, and SP.

The status bits also give advanced information concerning the next operating state of the MAC-8. Information (such as *HALT* resulting from a DMA request or HALT instruction, *TRAP* resulting from an illegal opcode or internal malfunction, and *opcode fetch*) can be decoded from the status signals.

5.1.5 Interrupt and Reset Timing

In most microcomputer systems, Interrupt and Reset requests are asynchronous events. The pending request usually is maintained until resolved by the CPU.

In the MAC-8, the timing for Interrupt ($\overline{\text{INT REQ}}$) and Reset ($\overline{\text{RESET}}$) requests is identical. The maximum setup time to ADDRESS change (T_{is} specification) is 80 ns, which means the strobe window for $\overline{\text{INT REQ}}$ or $\overline{\text{RESET}}$ (I/R) valid occurs toward the end of the clock cycle. If $\overline{\text{INT REQ}}$ or $\overline{\text{RESET}}$ is not latched, then the hold time from ADDRESS change (T_{ih} specification) is 150 ns maximum.

$\overline{\text{INT REQ}}$ or $\overline{\text{RESET}}$ is active when a logic zero level is applied to pin 24 or 25, respectively. The pending request is resolved when the MAC-8 performs a READ operation from address location[2] X(FFFF), thus indicating that internal latching of the request has been accomplished. Therefore, a $\overline{\text{READ}}$ signal and the decoded address X(FFFF) can be used to generate an acknowledge to clear requests.

One factor concerning Interrupt and Reset should be considered when designing the system. During the Interrupt and Reset cycle sequences, the all-ones address comes out on the address bus 120 ns after the high to low transition of CLK OUT. Therefore, the address bus starts changing much later in the cycle than normal. While neither a $\overline{\text{READ}}$ nor a $\overline{\text{WRITE}}$ signal is present, this delay can still affect the integrity of system memory (particularly dynamic memories).

5.2 Peripherals

A variety of peripheral devices are available for MAC-8 based microcomputer systems. Among them are the following peripherals which are described in this chapter:

- WE 28A Random Access Memory (RAM)
- WE 146D Peripheral Timer

---

[2] The notation *X(n)* denotes the hexadecimal number, *n*.

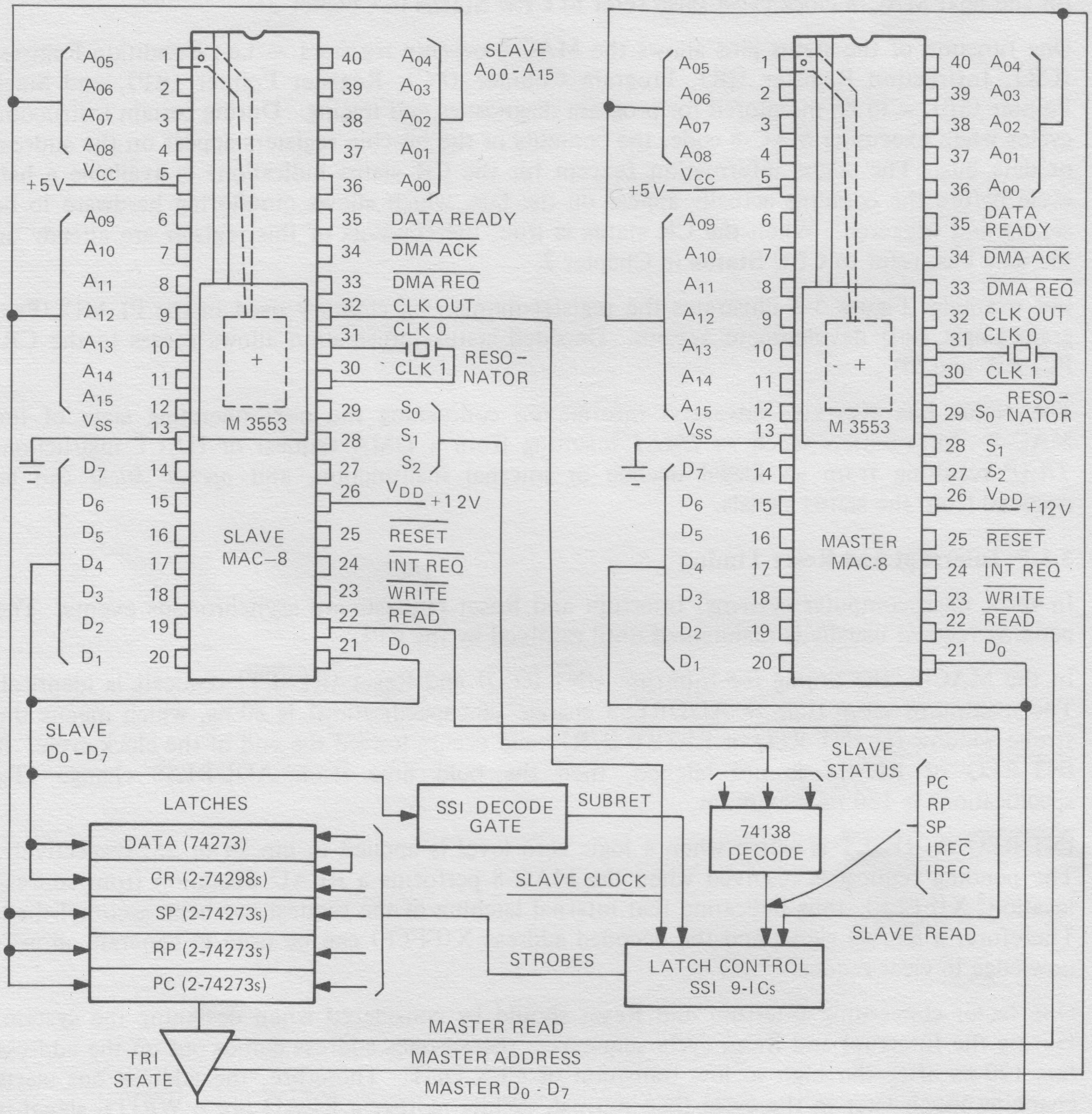

SLAVE
A00 - A15
A05
A06
A07
A08
VCC
+5V
A09
A10
A11
A12
A13
A14
A15
VSS
D7
D6
D5
D4
D3
D2
D1
A04
A03
A02
A01
A00
DATA READY
DMA ACK
DMA REQ
CLK OUT
CLK 0
CLK 1
RESO-
NATOR
S0
S1
S2
VDD
+12V
RESET
INT REQ
WRITE
READ
D0
M 3553
SLAVE
MAC-8
MASTER
MAC-8
SLAVE
D0 - D7
LATCHES
DATA (74273)
CR (2-74298s)
SP (2-74273s)
RP (2-74273s)
PC (2-74273s)
SSI DECODE
GATE
SUBRET
SLAVE
STATUS
74138
DECODE
PC
RP
SP
IRFC
IRFC
SLAVE CLOCK
SLAVE READ
STROBES
LATCH CONTROL
SSI 9-ICs
TRI
STATE
MASTER READ
MASTER ADDRESS
MASTER D0 - D7

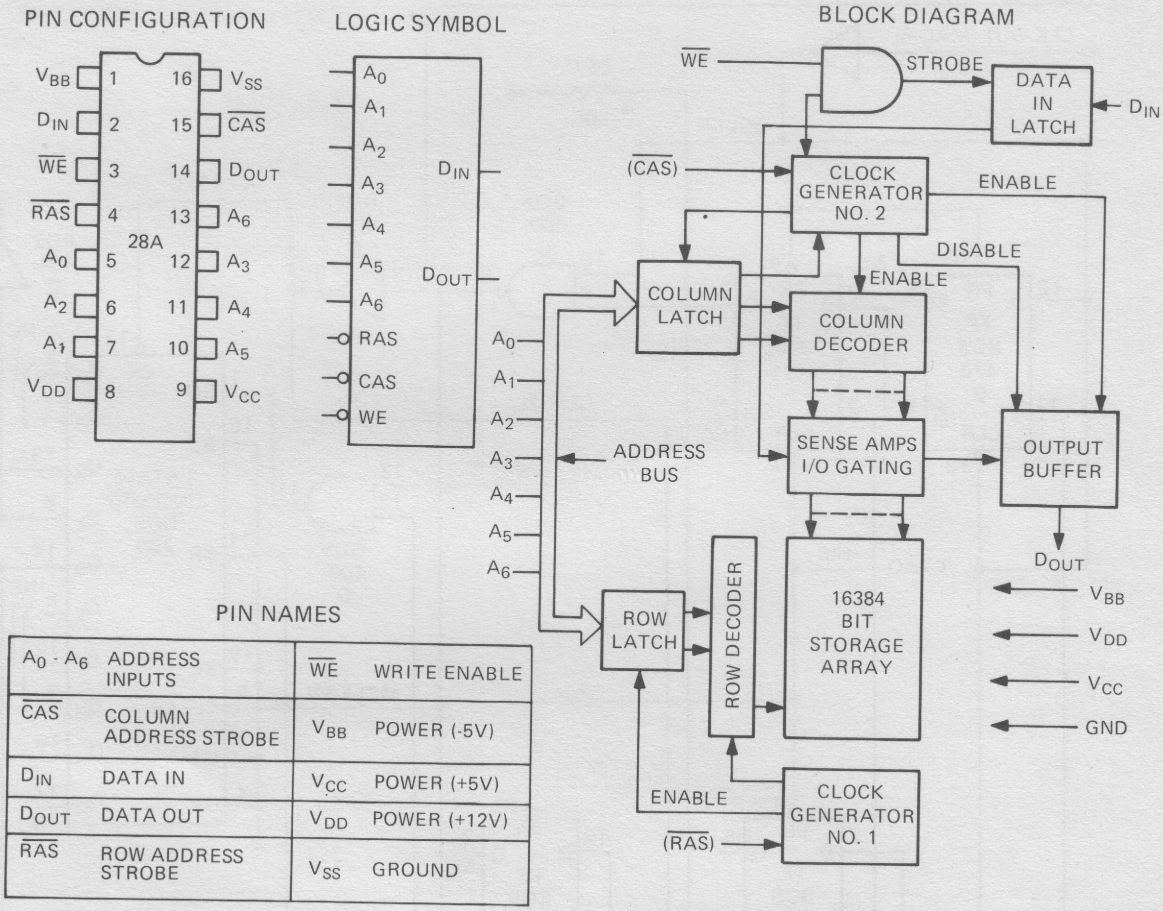
Figure 5-3. Snapshotting MAC-8 Registers

- Intel 8251/A Programmable Communication Interface
- Intel 8214 Priority Interrupt Controller
- Intel 8253 Programmable Interval Timer

The descriptions that follow illustrate the functional utility of these peripherals in MAC-8 based systems. The specific implementations discussed are, by no means, universally applicable — they represent one designer's solution to a particular application problem. However, they are indicative of the type of design considerations required in any microprocessor system design.

5.2.1 WE 28A RAM

Western Electric's 28A RAM is a 16Kx1 bits dynamic memory that is pin-for-pin compatible with the industry standard Intel 2117 RAM. Figure 5-4 is a block diagram of the 28A and illustrates its pin configuration.


PIN CONFIGURATION
V_{BB} 1
D_{IN} 2
$\overline{WE}$ 3
$\overline{RAS}$ 4
A_0 5
A_2 6
A_1 7
V_{DD} 8
28A
16 V_{SS}
15 $\overline{CAS}$
14 D_{OUT}
13 A_6
12 A_3
11 A_4
10 A_5
9 V_{CC}
LOGIC SYMBOL
A_0
A_1
A_2
A_3
A_4
A_5
A_6
RAS
CAS
WE
D_{IN}
D_{OUT}
BLOCK DIAGRAM
$\overline{WE}$
STROBE
DATA IN LATCH
D_{IN}
($\overline{CAS}$)
CLOCK GENERATOR NO. 2
ENABLE
DISABLE
ENABLE
COLUMN LATCH
COLUMN DECODER
A_0
A_1
A_2
A_3
A_4
A_5
A_6
ADDRESS BUS
SENSE AMPS I/O GATING
OUTPUT BUFFER
D_{OUT}
ROW LATCH
ROW DECODER
16384 BIT STORAGE ARRAY
V_{BB}
V_{DD}
V_{CC}
GND
ENABLE
($\overline{RAS}$)
CLOCK GENERATOR NO. 1
PIN NAMES

A_0 - A_6	ADDRESS INPUTS	$\overline{WE}$	WRITE ENABLE
$\overline{CAS}$	COLUMN ADDRESS STROBE	V_{BB}	POWER (-5V)
D_{IN}	DATA IN	V_{CC}	POWER (+5V)
D_{OUT}	DATA OUT	V_{DD}	POWER (+12V)
$\overline{RAS}$	ROW ADDRESS STROBE	V_{SS}	GROUND

Figure 5-4. 28A RAM Block Diagram

Address multiplexing and refresh must both be considered when using these RAMs in a system design.

Address Multiplexing

The 28A RAM has a 16-pin package configuration in which seven pins serve as address input lines. Thus, 14-bit address lines are multiplexed into these seven lines, designated A_0 through A_6. Two timing signals associated with multiplexing latch the low and high portions of the address information into the RAM. These signals are designated Row Address Select ($\overline{RAS}$) and Column Address Select ($\overline{CAS}$).

Multiplexing can be performed by direct use of transistor-transistor logic medium-scale integration — TTL(MSI) — circuitry or by means of a special-purpose integrated circuit, such as the Intel 3242, that is designed specifically for multiplexing.

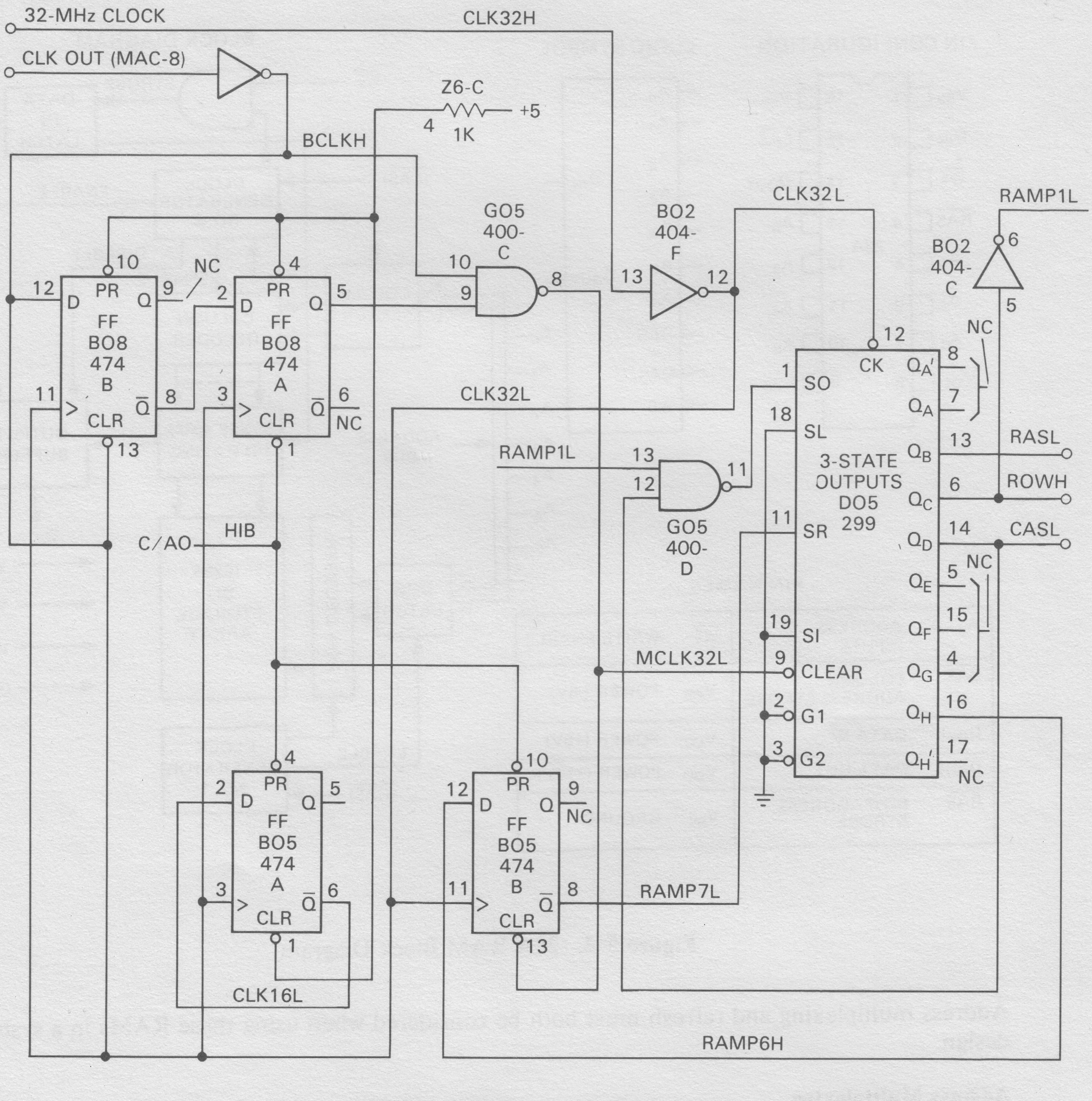

32-MHz CLOCK
CLK OUT (MAC-8)
CLK32H
Z6-C
+5
4
1K
BCLKH
GO5
400-
C
10
9
8
BO2
404-
F
13
12
CLK32L
RAMP1L
BO2
404-
C
6
5
NC
12
10
NC
9
2
4
5
PR
D
Q
FF
BO8
474
B
11
8
3
A
6
CLR
13
1
NC
CLK32L
RAMP1L
13
12
11
GO5
400-
D
CK
SO
SL
SR
SI
CLEAR
G1
G2
3-STATE
OUTPUTS
DO5
299
1
18
11
19
9
2
3
12
Q_A'
Q_A
Q_B
Q_C
Q_D
Q_E
Q_F
Q_G
Q_H
Q_H'
8
7
13
6
14
5
15
4
16
17
NC
RASL
ROWH
CASL
NC
MCLK32L
C/AO
HIB
4
2
5
FF
BO5
474
A
3
6
1
10
12
9
NC
FF
BO5
474
B
11
8
13
RAMP7L
CLK16L
RAMP6H

Figure 5-5. RAM Timing Signal Generator

The falling edge of the MAC-8's clock out (CLK OUT) signal can be used as the reference mark for generating timing signals. A 16-MHz or higher frequency oscillator can provide the necessary timing graduations within the MAC-8's 2-MHz cycle. Figure 5-5 illustrates a typical timing generator circuit using a 32-MHz clock to generate the $\overline{\text{RAS}}$, $\overline{\text{CAS}}$, and Row Enable (ROWENB) signals. The ROWENB signal controls the address multiplexer to select the low or high seven bits of the address.

Refresh

Dynamic memory uses a capacitance storage mechanism to retain data. Because the charge on such devices leaks, dynamic memory must be recharged periodically for the data to remain valid.

The address sequencing given to the RAM during the refresh cycle must be of a special nature; i.e., the 128 *row* address locations must be provided to the RAM every 2 ms. Several schemes exist for performing this task:

- The simplest technique is to introduce a wait state to the CPU at regular intervals and let a separate counter produce the 128-address sequence.
- A second technique, which is transparent to the MAC-8, is to split the memory into quadrants using the low-order address bits. When one quadrant gets selected, the other quadrant refreshes and *vice versa*; however, this technique requires memory sizes of 32K bytes or larger. Refreshing memory in this manner does not require additional wait states. When the CPU is in a *halt* state, the quadrants no longer toggle; hence, other provisions for refreshing must be inserted. The quadrant-segmenting approach needs separate counters and multiplexers for each quadrant.
- Another technique, which also is transparent to the MAC-8, involves sampling the 8-bit opcode for each instruction being executed. All MAC-8 instructions have at least one machine cycle that does not access memory. Through use of a table look-up ROM, the exact cycle can be determined and used for refreshing RAM. This technique can be applied equally well to memory sizes of 16K or larger. Large memories require only one refresh counter.

5.2.2 WE 146D Peripheral Timer

Western Electric's 146D peripheral timer functions as the MAC-8 peripheral interface circuit. This 18-pin DIP permits the MAC-8 to be used with a variety of commercial I/O devices, which include the Intel 8251 through 8259 and the AMD 9517 through 9519 series of peripherals. The setup time, hold time, and pulse width requirements for READ and WRITE signals to these devices do not allow direct interfacing with the MAC-8 microprocessor.

To partially solve this compatibility problem, the 146D generates a single wait state for each access and positions the READ and WRITE signals over two clock cycles, as shown in Figure 5-6. The regenerated signals meet the timing requirements[3] of these commercial I/O devices. Inputs to 146D pins CLK OUT, RD CLK, and WR CLK are the CLK OUT, $\overline{\text{READ}}$, and $\overline{\text{WRITE}}$ signals from the MAC-8, respectively. Outputs from the 146D are DATA READY to the MAC-8 and $\overline{\text{RD}}$ and $\overline{\text{WR}}$ to the peripherals. The 146D can handle up to four peripherals, selected by the S_0 and S_1 address inputs. Inputs S_2, S_3, $\overline{S_4}$, and $\overline{S_5}$ provide address decoding to select the circuit.

The 146D also can be used as a wait-state generator to provide a single wait state per access for memory and other devices where MAC-8 compatibility is not a problem. For this application, the $\overline{\text{READ}}$ and $\overline{\text{WRITE}}$ signals from the MAC-8 can be used while $\overline{\text{RD}}$ and $\overline{\text{WR}}$ from the 146D are left unconnected.

Another application of the 146D is to generate a wait state for every access at any address. The MAC-8 ignores the DATA READY signal other than during clock cycles when a $\overline{\text{READ}}$ or $\overline{\text{WRITE}}$ signal is active (low logic level). Therefore, the DATA READY signal can be held low continuously until a READ or a WRITE occurs. At that time, after one wait state, the DATA

---

[3] Certain devices, such as the Intel 8251 and 8257, require further preconditioning of control signals.

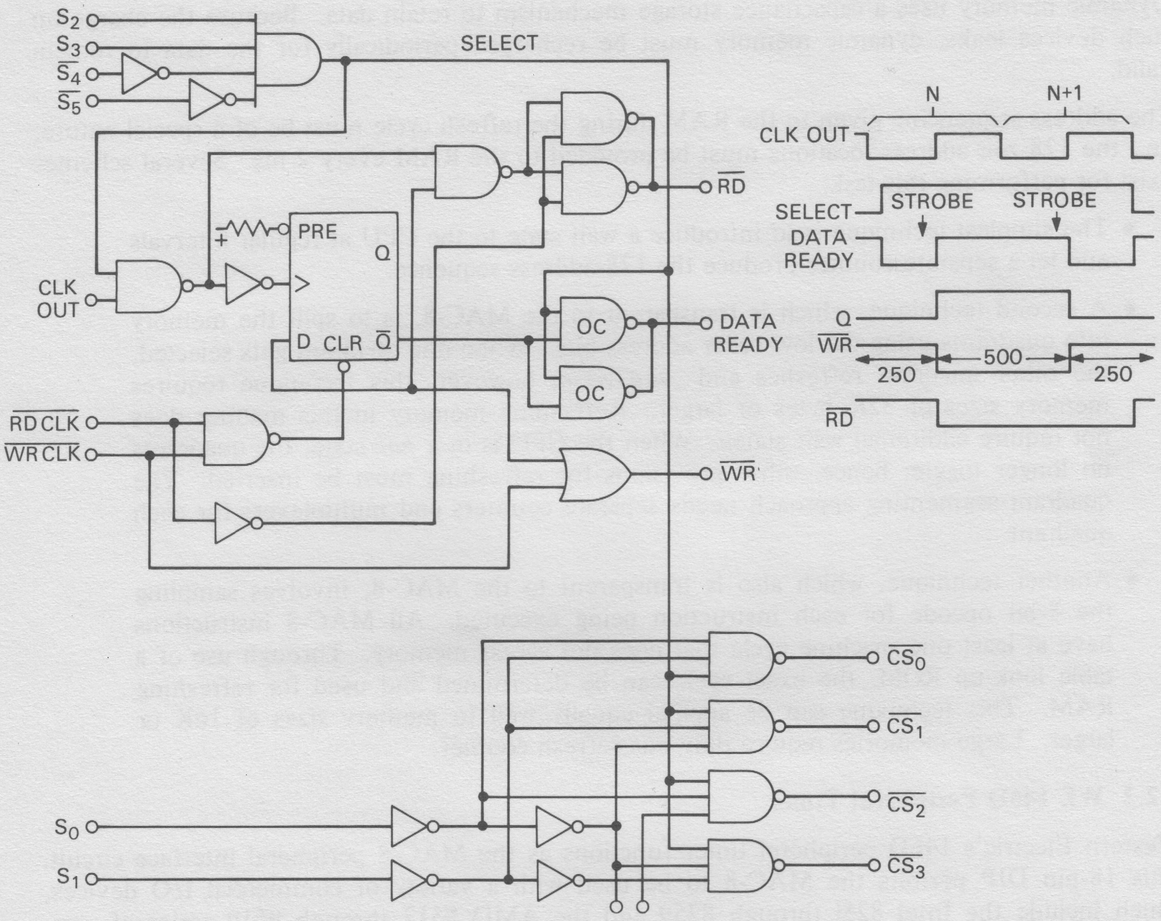

S2
S3
S4
S5
SELECT
N
N+1
CLK OUT
RD
SELECT
STROBE
STROBE
DATA READY
PRE
Q
CLK OUT
D
CLR
Q
OC
OC
DATA READY
Q
WR
250
500
250
RD
RD CLK
WR CLK
WR
CS0
CS1
CS2
CS3
S0
S1

Figure 5-6. WE 146D Peripheral Timer

READY signal can be made high to permit the READ or WRITE access and then made low again. To accomplish this *reverse* wait state generator sequence with the 146D, the S_2 through $\overline{S_5}$ inputs must be permanently enabled.

5.2.3 Intel 8251/A Programmable Communication Interface

The Intel 8251/A implements USART (Universal Synchronous/Asynchronous Receiver/Transmitter) operation with the MAC-8 processor. This device converts outgoing parallel-format system data into a continuous serial-format data stream for transmission. Simultaneously, it can convert incoming serial data streams into parallel system data for reception.

Figure 5-7 illustrates typical connections from the Intel 8251/A to the MAC-8; refer to the 8251/A specification sheet for additional details:

- The input signal to $\overline{\text{CTS}}$ (pin 17) must be low when the 8251/A chip is to transmit.
- The output signal from TxD (pin 19) is TTL compatible and normally high in an idle state. The signal undergoes one inversion and conversion (via the transmitter

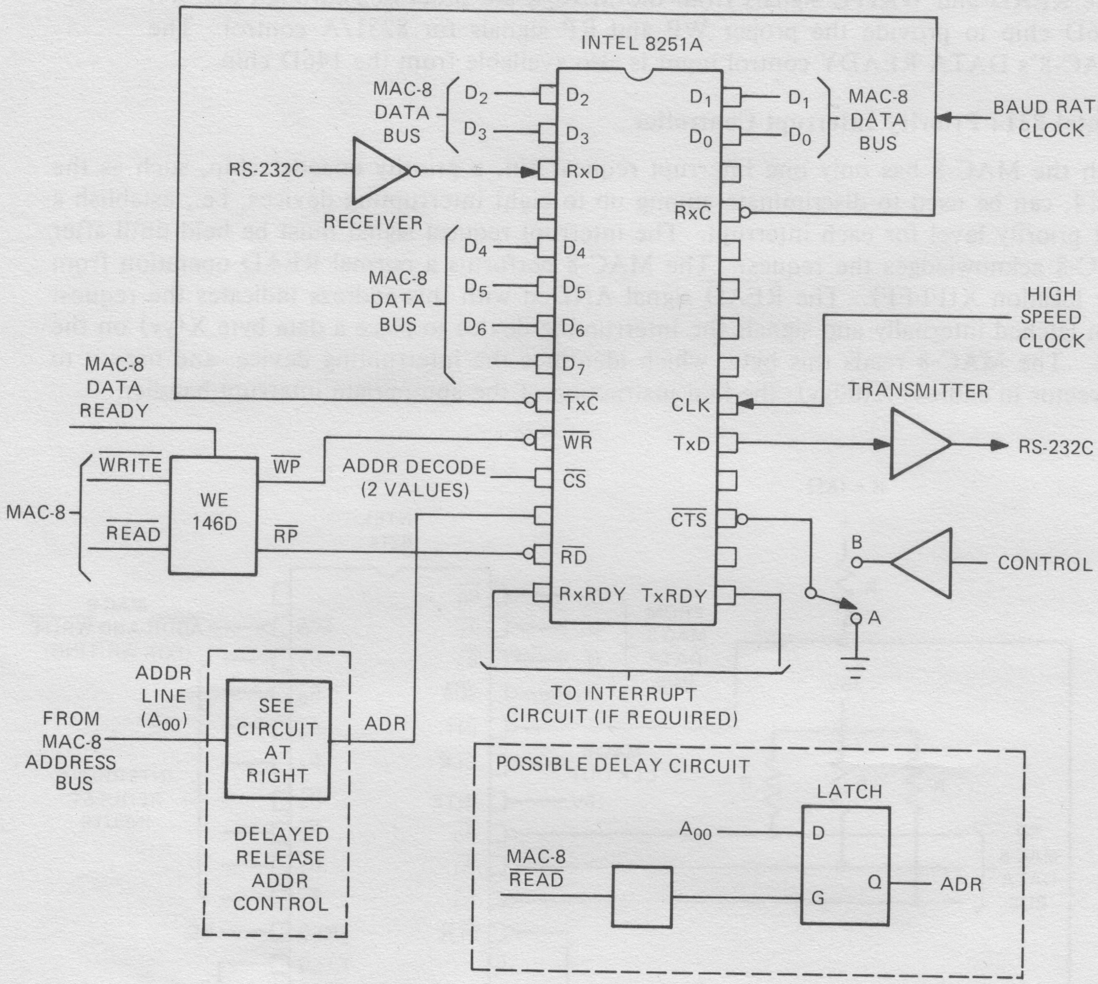

INTEL 8251A
MAC-8 DATA BUS
D2
D3
D2
D3
D1
D0
D1
D0
MAC-8 DATA BUS
BAUD RATE CLOCK
RS-232C
RxD
RECEIVER
RxC
MAC-8 DATA BUS
D4
D5
D6
D7
HIGH SPEED CLOCK
MAC-8 DATA READY
TxC
CLK
TRANSMITTER
WRITE
WP
WR
TxD
RS-232C
MAC-8
WE 146D
ADDR DECODE (2 VALUES)
CS
READ
RP
CTS
RD
B
CONTROL
RxRDY
TxRDY
A
TO INTERRUPT CIRCUIT (IF REQUIRED)
ADDR LINE (A00)
FROM MAC-8 ADDRESS BUS
SEE CIRCUIT AT RIGHT
ADR
DELAYED RELEASE ADDR CONTROL
POSSIBLE DELAY CIRCUIT
LATCH
A00
D
MAC-8 READ
G
Q
ADR

Figure 5-7. Intel 8251/A USART Operation

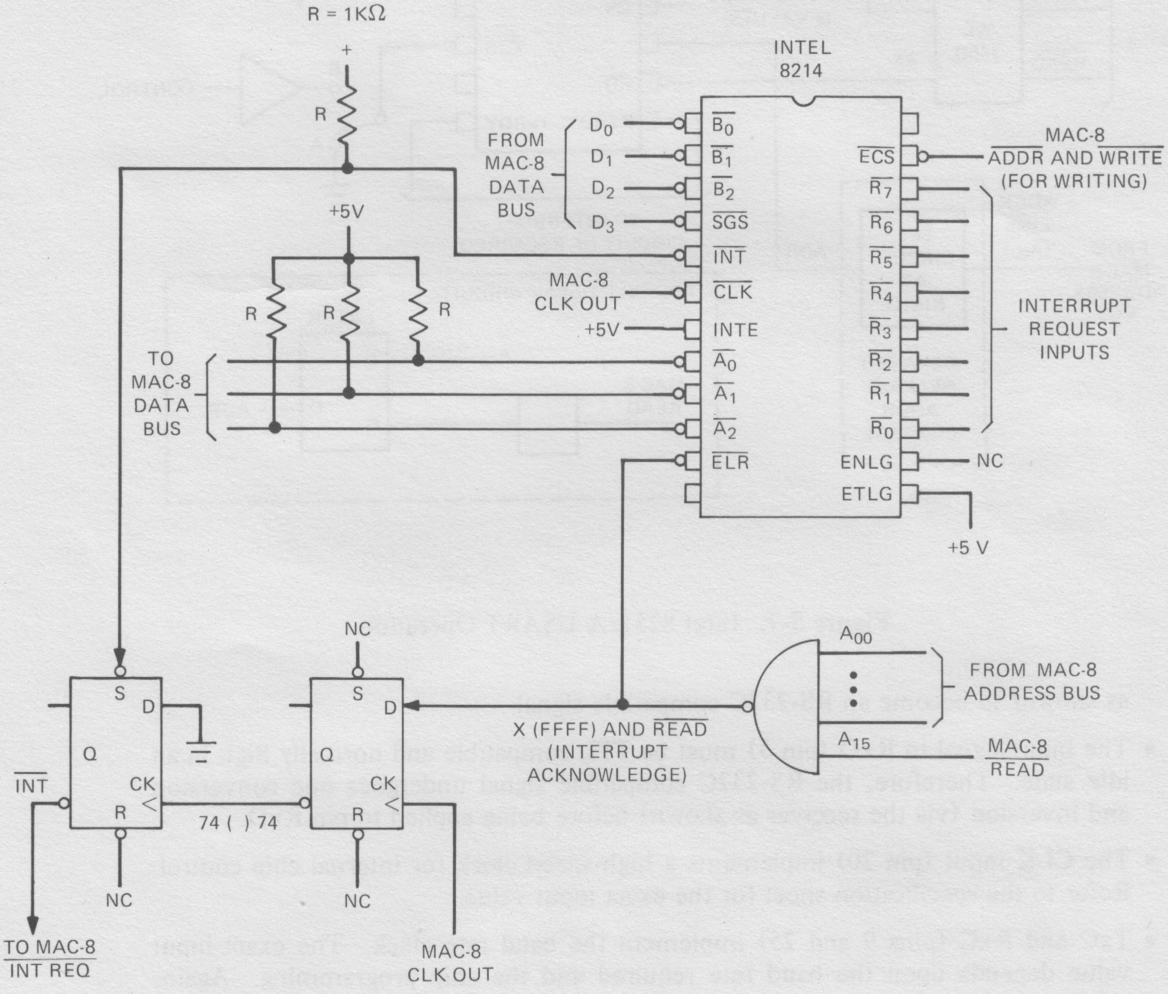
as shown) to become an RS-232C compatible signal.

- The input signal to RxD (pin 3) must be TTL compatible and normally high in an idle state. Therefore, the RS-232C compatible signal undergoes one conversion and inversion (via the receiver as shown) before being applied to pin RxD.
- The CLK input (pin 20) implements a high-speed clock for internal chip control. Refer to the specification sheet for the exact input value.
- TxC and RxC (pins 9 and 25) implement the baud rate clock. The exact input value depends upon the baud rate required and the chip programming. Again, refer to the specification sheet for details.
- The C/$\overline{D}$ line (pin 12) connects to a delayed release address control circuit to prevent destroying the receiver ready flag setting. Figure 5-7 also shows a configuration that could be used for this delay circuit.

- The $\overline{\text{READ}}$ and $\overline{\text{WRITE}}$ signals from the MAC-8 are processed through the WE 146D chip to provide the proper $\overline{\text{WP}}$ and $\overline{\text{RP}}$ signals for 8251/A control. The MAC-8's DATA READY control input is also available from the 146D chip.

5.2.4 Intel 8214 Priority Interrupt Controller

Although the MAC-8 has only one interrupt request pin, a priority encoder chip, such as the Intel 8214, can be used to discriminate among up to eight interrupting devices, i.e., establish a different priority level for each interrupt. The interrupt request signal must be held until after the MAC-8 acknowledges the request. The MAC-8 performs a normal READ operation from memory location X(FFFF). The $\overline{\text{READ}}$ signal ANDed with this address indicates the request has been latched internally and signals the interrupting device to place a data byte X(yy) on the data bus. The MAC-8 reads this byte, which identifies the interrupting device, and uses it to form a vector to address X(00yy), the first instruction of the appropriate interrupt handler.



R = 1KΩ
+
R
+5V
R
R
R
TO MAC-8 DATA BUS
FROM MAC-8 DATA BUS
D0
D1
D2
D3
MAC-8 CLK OUT
+5V
INTEL 8214
B0
B1
B2
SGS
INT
CLK
INTE
A0
A1
A2
ELR
ECS
R7
R6
R5
R4
R3
R2
R1
R0
ENLG
ETLG
MAC-8 ADDR AND WRITE (FOR WRITING)
INTERRUPT REQUEST INPUTS
NC
+5 V
NC
S
D
Q
CK
R
S
D
Q
CK
R
INT
74 ( ) 74
NC
NC
TO MAC-8 INT REQ
MAC-8 CLK OUT
X (FFFF) AND READ (INTERRUPT ACKNOWLEDGE)
A00
A15
FROM MAC-8 ADDRESS BUS
MAC-8 READ

Figure 5-8. Encoder Chip Connections

Figure 5-8 illustrates the connections from the 8214 to the MAC-8; refer to the Intel 8214 specification sheet for additional details:

- The MAC-8 CLK OUT signal implements an input clock when applied to the 8214 $\overline{\text{CLK}}$ (pin 6).
- The 8214 can handle interrupt requests from eight external devices. The interrupt request input is an active (low logic level) signal applied to one of the request levels $\overline{R}_0$ through $\overline{R}_7$ (pins 15 through 22).
- The control word is written into the 8214 chip as an active (low logic level) signal to the 8214's pins $\overline{B}_0$, $\overline{B}_1$, $\overline{B}_2$, and $\overline{\text{SGS}}$ (pins 1 through 4) from the MAC-8's data bus pins D_0 through D_3, respectively.
- An active (low logic level) signal to $\overline{\text{ECS}}$ (pin 22) provides input control for writing the 8214 chip. The input signal is an address ($\overline{\text{ADDR}}$) ANDed with an active (low logic level) $\overline{\text{WRITE}}$ output signal from the MAC-8 after data is stable.
- The interrupt acknowledge from the MAC-8 provides an active (low logic level) input signal to the 8214 $\overline{\text{ELR}}$ (pin 11), which is the control for the output vector onto the MAC-8's data bus. This pin also resets the input gate in the 8214. The input signal to this pin is the interrupt acknowledge, i.e., the address X(FFFF) ANDed with the $\overline{\text{READ}}$ signal from the MAC-8.
- Outputs from the 8214's $\overline{A}_0$, $\overline{A}_1$, and $\overline{A}_2$ (pins 8 through 10) are connected to the MAC-8 data bus to generate the output vector. Because these outputs are open collector, pull-up resistors must be used.
- The active (low logic level) output signal (which is open collector) from the 8214 $\overline{\text{INT}}$ (pin 5) provides the active (low logic level) input signal to the MAC-8 $\overline{\text{INT REQ}}$ pin (also open collector). The $\overline{\text{INT}}$ output lasts a maximum of one MAC-8 clock cycle. The request is latched into the second flip-flop (as shown in the illustration); output from this flip-flop connects to the MAC-8 $\overline{\text{INT REQ}}$ pin. The interrupt request must terminate before the end of the MAC-8 clock cycle in which the interrupt acknowledge was generated. The first flip-flop shown in the figure causes this request termination to take place when the vector (on the MAC-8 data bus) is being read.

5.2.5 Intel 8253 Programmable Interval Timer

The Intel 8253 programmable timer can be connected to a MAC-8 chip (as shown in Figure 5-9) for operation at 2 MHz. As illustrated, Western Electric's 146D peripheral timer is used as the interface circuit. The setup time, hold time, and pulse width requirements for the 8253's read and write signals, $\overline{\text{RD}}$ and $\overline{\text{WR}}$, do not allow direct interfacing with the corresponding MAC-8 $\overline{\text{READ}}$ and $\overline{\text{WRITE}}$ signals.

The 146D, when connected between the MAC-8 and 8253 as shown in the figure, generates a wait state within the MAC-8, thus ensuring minimum pulse width requirements of the 8253. In particular, the inherent delay (READ) and circuit operation (WRITE) of the 146D satisfy the address hold time requirements for READ and WRITE. Figure 5-10 illustrates the timing at 2 MHz for READ and WRITE operations using the WE 146D connected between the MAC-8 and the Intel 8253.

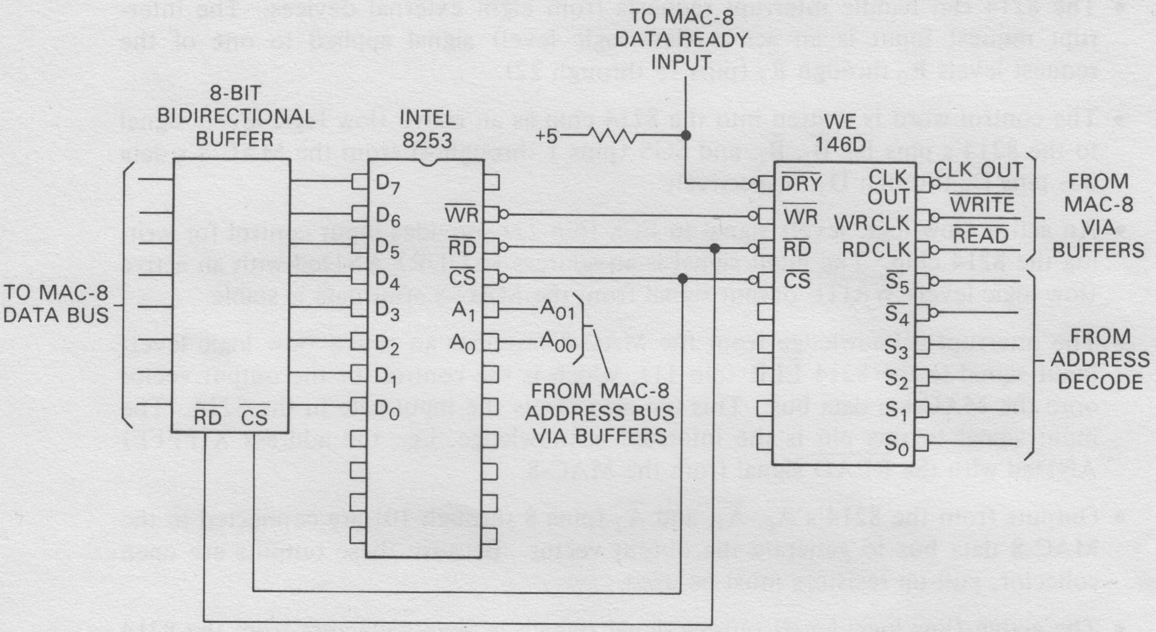

TO MAC-8
DATA READY
INPUT
8-BIT
BIDIRECTIONAL
BUFFER
INTEL
8253
+5
WE
146D
TO MAC-8
DATA BUS
D7
D6
D5
D4
D3
D2
D1
D0
RD
CS
WR
RD
CS
A1
A0
A01
A00
FROM MAC-8
ADDRESS BUS
VIA BUFFERS
DRY
WR
RD
CS
CLK
OUT
WR CLK
RD CLK
S5
S4
S3
S2
S1
S0
CLK OUT
WRITE
READ
FROM
MAC-8
VIA
BUFFERS
FROM
ADDRESS
DECODE

Figure 5-9. Intel 8253 Connections to MAC-8
(via WE 146D Peripheral Timer)

5.3 System Integration

Several MAC-8 systems have been designed and proven in. They range from small, minimal DIP count systems (e.g., a MAC-8 with an ROM-RAM-I/O device) to large, sophisticated systems (e.g., PLAID — Programming Logic Aid).

5.3.1 Automated Coin Toll Service (ACTS)

The ACTS system uses a MAC-8 in conjunction with an ROM-RAM-I/O device to identify coin deposit signals from a coin station. Figure 5-11 is a simplified block diagram of this two-chip microcomputer.

A MAC-8 microprocessor is interfaced to an MCS-6530[4] ROM-RAM-I/O device by means of a few transistor-transistor logic (TTL) gates. The MCS-6530 chip utilizes MOS technology and contains 1K bytes of ROM, 64 bytes of RAM, and two 8-bit I/O ports. The microprocessor system also contains a one-shot multivibrator as a sanity-detect circuit. When this circuit is not primed regularly, it resets the MAC-8 microprocessor.

---

[4] MCS is a registered trademark of MOS Technology Inc.

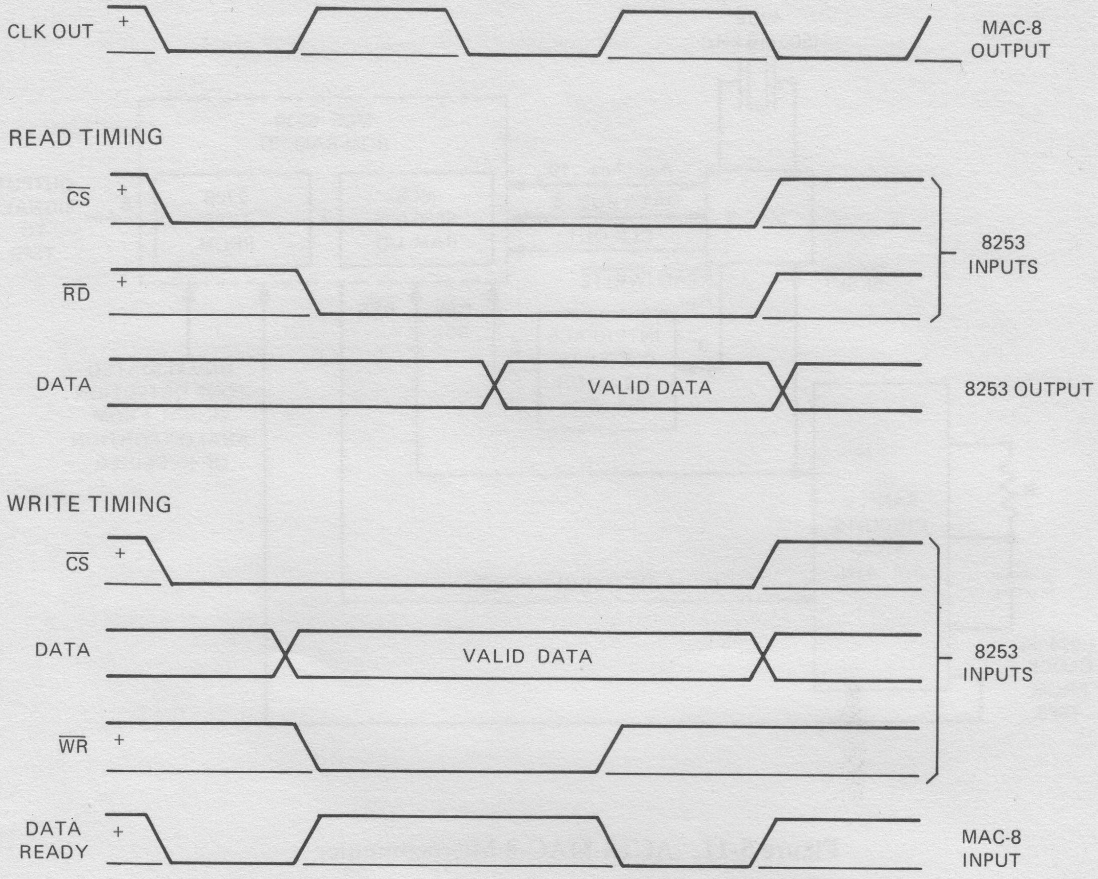

CLK OUT
+
MAC-8 OUTPUT
READ TIMING
CS
+
RD
+
8253 INPUTS
DATA
VALID DATA
8253 OUTPUT
WRITE TIMING
CS
+
DATA
VALID DATA
8253 INPUTS
WR
+
DATA READY
+
MAC-8 INPUT

Figure 5-10. MAC-8/Intel 8253 Timing at 2 MHz using a WE 146D

5.3.2 Programmed E-Telemetry (PROETEL) System

The PROETEL system (illustrated in Figure 5-12) is a Western Electric manufactured, general-purpose microcomputer designed to perform the common control functions for Programmed E-Telemetry terminals. The system was designed originally around the Intel 8080 CPU chip and required two WE JW-size cards for the CPU, buffers, and the necessary timing signals. A single JW-size MAC-8 card has been designed and proven in. This MAC-8 card replaces the two CPU cards required with the Intel chip.

In addition to these CPU control cards, Western Electric manufactures a series of standard PROETEL serial and parallel I/O cards, several sizes of RAM cards, and PROM/ROM cards. Western Electric also offers various sizes of apparatus mountings (card baskets) with common 3-layer printed circuit backplanes to connect the cards.

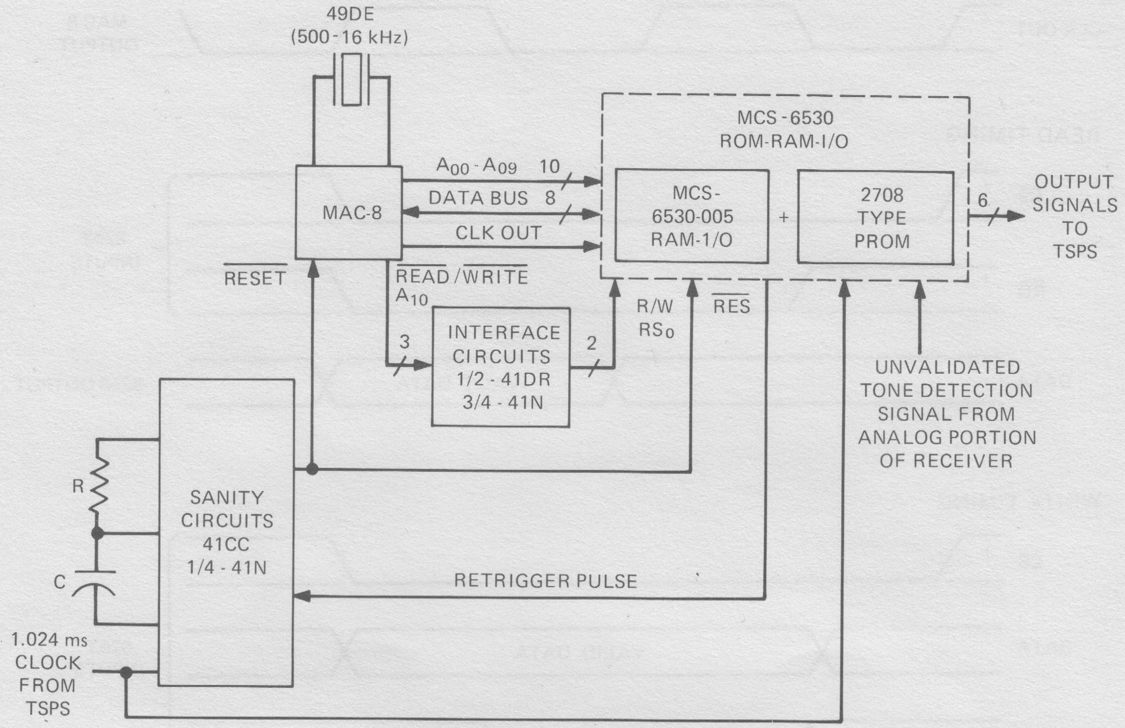

49DE
(500-16 kHz)
MAC-8
A00 - A09 10
DATA BUS 8
CLK OUT
READ/WRITE
A10
RESET
MCS-6530
ROM-RAM-I/O
MCS-
6530-005
RAM-1/O
+
2708
TYPE
PROM
6
OUTPUT
SIGNALS
TO
TSPS
R/W
RS0
RES
3
INTERFACE
CIRCUITS
1/2 - 41DR
3/4 - 41N
2
UNVALIDATED
TONE DETECTION
SIGNAL FROM
ANALOG PORTION
OF RECEIVER
R
C
SANITY
CIRCUITS
41CC
1/4 - 41N
RETRIGGER PULSE
1.024 ms
CLOCK
FROM
TSPS

Figure 5-11. ACTS MAC-8 Microcomputer

5.3.3 PLAID

PLAID (shown in Figure 5-13) is a sophisticated hardware/software development system that serves as a primary development tool for any MAC-8 based application.

The PLAID hardware[5] consists of three major building blocks:

- A MAC-8 master control system whose major function is to closely monitor and control the slave/user's MAC-8 system
- The MAC-8 slave system consisting of a MAC-8, 65K bytes of RAM with a 16K PROM overlay, and the MAC Cable interface to the user's actual hardware
- The monitor system which keeps track of the actions of the slave/user system.

The PLAID software[6] consists of the Satellite Processing System (SPS) and the symbolic debugger (m8db). The PLAID system interacts with the user via a terminal and with the UNIX system via a modem.

---

[5] Also refer to the *PLAID Hardware Reference Manual* (to be published).

[6] Also refer to the *PLAID Programmer's Manual* (to be published).

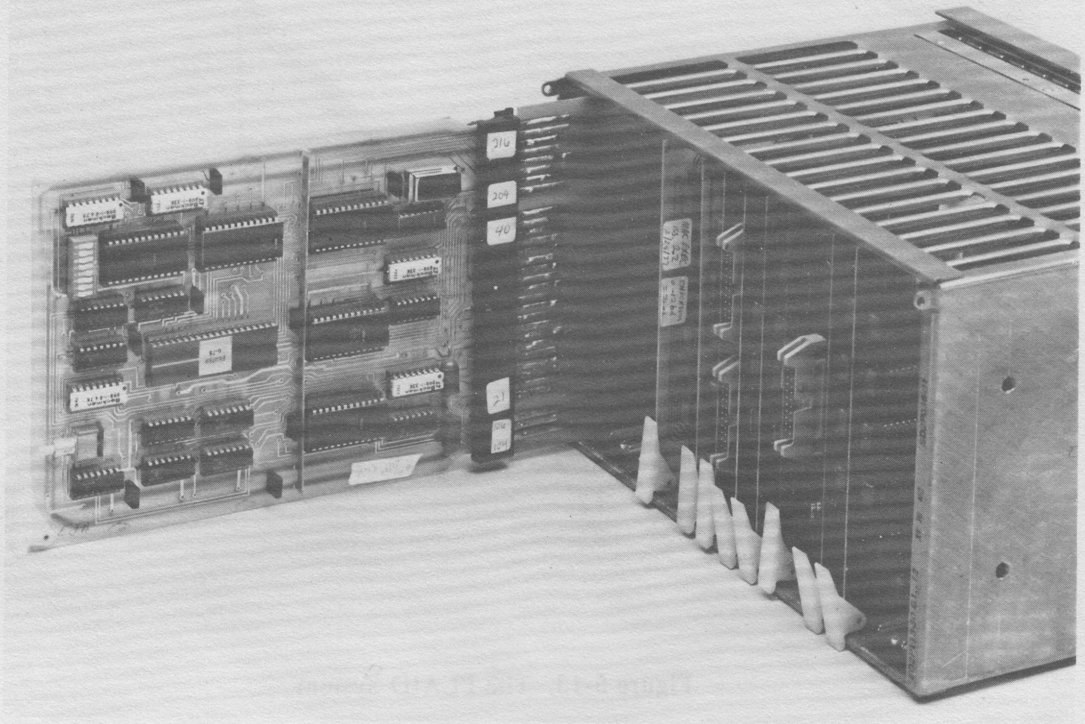


Figure 5-12. MAC-8/PROETEL System

5.3.4 MAC-8 MDS CPU Module

The Intellec[7] Microcomputer Development System (MDS) is designed to support the development and implementation of Intel Corporation's line of microcomputer systems (e.g., the MCS-48, MCS-80, and MCS-80 systems). The minimum system hardware consists of

- a standardized bus
- an 8-level maskable priority interrupt structure
- a real-time clock
- a 256-byte bootstrap loader
- a 2K-byte ROM-based monitor
- a 16K-byte RAM memory
- interfaces for teletype, CRT, high-speed paper-tape reader and/or punch, line printer, and PROM programmer.

The system resources can be expanded with optional RAM, ROM, and I/O modules.

---

[7] Registered trademark of Intel Corporation.

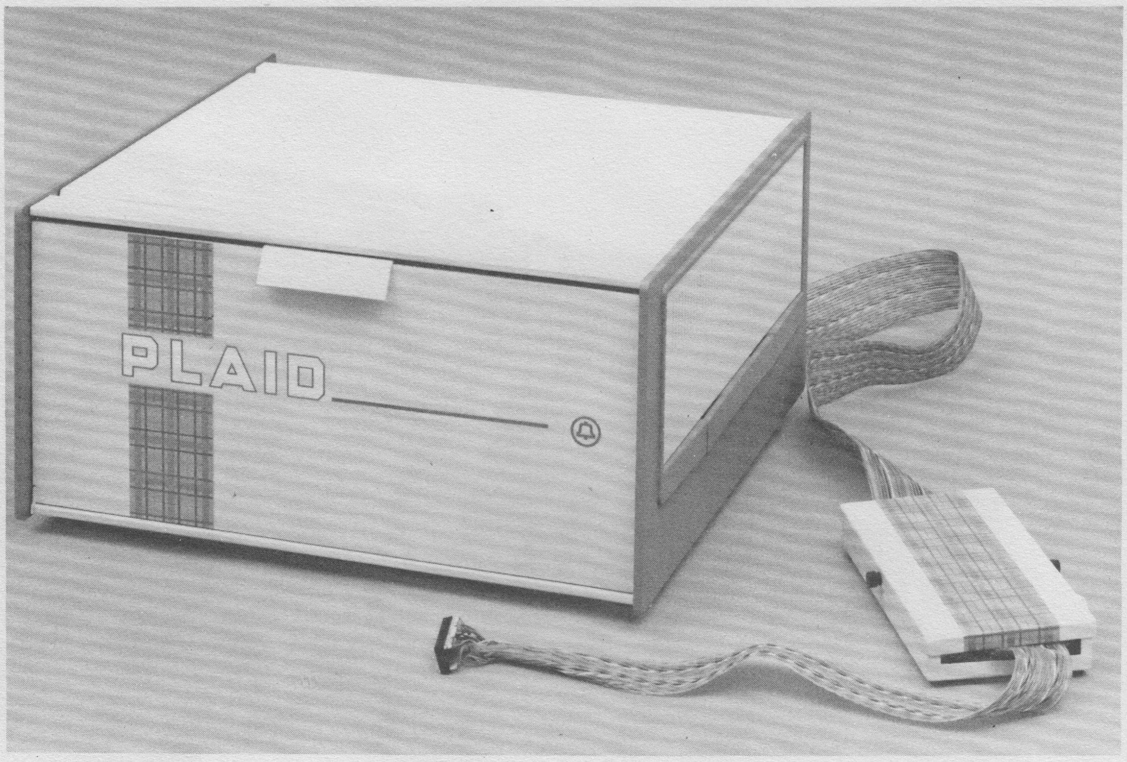


Figure 5-13. The PLAID System

The MAC-8 MDS CPU module retrofits the MAC-8 microprocessor to the MDS environment. When it is installed as shown in Figure 5-14, this CPU module allows the user to program in MAC-8 machine code while utilizing the MDS hardware resources.

The standard software package for MAC-8 retrofit systems is the MAC-EXEC system monitor, which is available in two forms. It provides the fundamental utility functions for the MDS. These functions include

- program loading
- memory display
- memory modification
- PROM programming
- program checkout
- debugging facilities and generalized I/O routines that can be shared by user's programs.

The system monitor can reside permanently within the system on PROM (either 1702 or 2708 PROM modules). Alternatively, it can be downloaded into RAM from the UNIX system using the bootstrap loader PROM (MAC-BOOT).

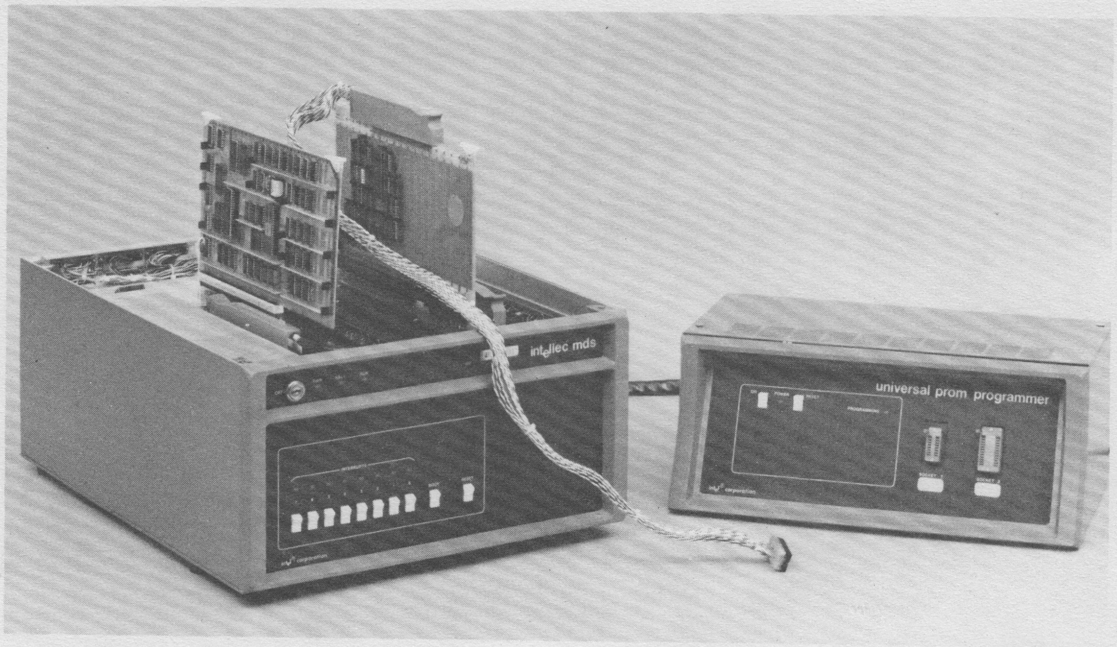

intellec mds
universal prom programmer

Figure 5-14. MAC-8 MDS CPU Module

Chapter 6

APPENDIX – REFERENCE TABLES

6. APPENDIX – REFERENCE TABLES

This section contains summary data and quick-reference tables for the following:

1. Assembly language tokens
2. Opcode assignments
3. Addressing modes
4. Instruction length

6.1 Assembly Language Tokens

The lowest identifiable unit of information in a language is called a *token.* The tokens for the MAC-8 assembler are:

Keywords	Words that have special meanings for the assembler
Identifiers	Symbolic names of memory locations
Constants	Fixed quantities used in assembly language statements
Operators	Symbols that define operations to be performed
Punctuation	
Blanks and Tabs	
Comments	

Because this manual is intended primarily for hardware designers, a detailed discussion of the tokens is not included here. Refer to the *MAC-8 Programmer's Manual* for the lexical conventions governing their usage.

6.1.1 Keywords

Listed below are the keywords of the MAC-8 assembler together with their reserved definitions.

Register Names

These keywords identify MAC-8 registers which can be specified in assembly language statements.

a0, ..., a15	A Registers 0 through 15; contain 8-bit data
b0, ..., b15	B Registers 0 through 15; contain 16-bit data or pointers to memory locations
cr	Condition Register
pc	Program Counter
rp	Register Pointer

sp	Stack Pointer

Condition Names

The column at the left lists the *true* condition name. To specify a *false* condition name, use the indicated name prefixed by the *not* operator (!) — e.g., *!neg* specifies the false condition, *not negative result*. A true condition is satisfied when the appropriate condition bit is set (1); a false condition is satisfied when the bit is clear (0).

neg	N; negative result; duplicates sign bit
zero	Z; zero result
ovfl	V; arithmetic overflow
carry	C; carry or borrow
ones	A; all ones in result
odd	Odd result; duplicates low bit
enable	Interrupts enabled
flag	User-defined condition
lt	N^V; arithmetically less than zero
lteq	(N^V)|Z; *arithmetically* less than or equal to zero
llteq	C|Z; *logically* less than or equal to zero
homog	A|Z; logically homogeneous; all ones or all zeros
shovfl	N^C; overflow on arithmetic left shift
always	Always set; unconditional jump, call, or return

Operator Names

These keywords identify operations to be performed.

bit	Logical operator for branch-on-bit instruction
bitsum	Count number of ones
brto	Short jump
byte	Assemble 8-bit constant
call	Call (execute) subroutine
clear	Clear condition(s)
extend	Extend sign
flo	Find leftmost one
floc	Find and clear leftmost one
halt	Put processor in HALT state
goto	Jump
ireturn	Return from interrupt
logical	Treat as 8-bit data rather than signed 7-bit data

nop	No operation
pop	Pop from stack
push	Push onto stack
return	Return from subroutine
set	Set condition(s)
swap	Swap bytes in 16-bit word
test	Test under mask
word	Assemble 16-bit constant

Other Reserved Words

The following keywords are included here for information only. Refer to the *MAC-8 Programmer's Manual* for descriptions and usage.

break	Escape from a looping construct or switch
case	Part of *switch* construct
char	Declare and/or define identifier to be 8 bits
continue	Continue a loop
default	Part of *switch* construct
do	*DO* loop
else	*If-else* construct
extern	Declare identifier to be externally defined
for	*For* loop
if	Conditional (branch, call, return)
int	Declare and/or define identifier to be 16 bits
static	Declare and/or define identifier to be local
switch	*Switch* high-level construct
union	Declare and/or define identifiers that occupy common memory
while	*While* loop

The following keywords are reserved words in the C language and are being reserved in the MAC-8 assembler for future implementation.

auto
double
entry
float
register
sizeof
struct

6.1.2 Identifiers

An identifier consists of a sequence of alphanumeric characters (lower case or UPPER CASE, including underscore) that symbolically names a memory location. The first character may not be numeric and only the first 8 characters are significant. *Keywords may not be used as identifiers.*

An identifier that refers to an absolute memory location consists of an asterisk followed by a constant, the absolute memory location; for example, the identifier *077760 uses an octal constant (see **Constants**) to identify an absolute memory location.

Refer to the *MAC-8 Programmer's Manual* for additional details regarding identifiers.

6.1.3 Constants

A constant is a fixed quantity used in an assembly language statement. Constants may take any of the following forms; refer to the *MAC-8 Programmer's Manual* for additional details.

Constant Values

The magnitude of these constants may be represented in 4, 8, or 16 bits as determined by the context:

Octal	A sequence of digits (0 through 9); leading digit must be a zero; digits 8 and 9 have the octal values 10 and 11, respectively
Decimal	A sequence of digits (0 through 9); leading digit must *not* be zero
Hexadecimal	A sequence of digits (0 through 9) and the letters a through f; prefixed with 0x
Character	An ASCII character enclosed in single quotes, e.g., ′x′; the value of this constant is the value of that character in the character set

Constant Expressions

A constant expression consists of constants, operators, and parentheses. Constant expressions may be used anywhere a constant is valid.

String Constants

A string constant consists of a sequence of ASCII characters enclosed in double quotes (" "). String constants may only be used to initialize character arrays (refer to the *MAC-8 Programmer's Manual*).

6.1.4 Operators

The following operators are available to the MAC-8 assembler. Each is associated with either specific instructions, the addressing modes, or constant expressions. Some of the operators identified in this section were not discussed in this manual because their usages are beyond its scope; refer to the *MAC-8 Programmer's Manual* for complete descriptions.

Unary Operators

These tokens define operations on a single expression written adjacent to the operator, i.e., expression *operator*, or *operator* expression.

Unary Operator	Function
*	Indirection; thing pointed to by *expression*
&	Address of *expression*
−	Negate
~ or ¬	Complement (ASCII tilde or EBCDIC hook)
!	Not
++	Increment by 1
−−	Decrement by 1

Binary Operators

These tokens define operations to be performed on two expressions; i.e.,

expression *operator* expression

The operation generally changes the value of one of the operands. Assignment statements of the form

destination = destination *operator* source;

may also be written

destination =*operator* source;

where =*operator* is a token of the language, called a *compound operator.* No blanks may appear within tokens, but blanks should separate adjacent tokens.

Binary Operator	Compound Operator	Function
=	none	Replace
+	=+	Add
−	=−	Subtract
*	=*	Arithmetic shift left
/	=/	Arithmetic shift right
&	=&	AND
|	=|	OR
^	=^	EXCLUSIVE OR
<<	=<<	Shift left
>>	=>>	Shift right
<<<	=<<<	Rotate left
>>>	=>>>	Rotate right
$<<	=$<<	Rotate left through carry
>>$	=>>$	Rotate right through carry

Relational Operators

These tokens define relational operations between two expressions and are written as

expression *operator* expression

These operators are included for information only; refer to the *MAC-8 Programmer's Manual.*

Relational Operator	Function
==	Equal
=	Not equal
<	Less than
<=	Less than or equal
>	Greater than
>=	Greater than or equal

These operations yield 0 if the relation is false and 1 if it is true; neither operand is altered.

The following relational operators are reserved in the C language and are being reserved in the MAC-8 assembler for future implementation.

=%
&&
?
->
|

6.1.5 Punctuation

The following tokens are used as punctuation:

,	Comma
;	Semicolon
:	Colon
′	Single quote
"	Double quote
( )	Parentheses
{ }	Braces
[ ]	Brackets

6.1.6 Blanks and Tabs

A blank or tab must separate adjacent keywords, constants, or identifiers that are not separated otherwise. *Blank and tab characters may not be used within tokens.*

6.1.7 Comments

A comment begins with /* and ends with */; when the comment spans several lines, the intermediate lines must begin with *. A comment may be used anywhere a blank is legal. The assembler ignores comments.

6.2 Opcode Assignments

The table lists the function assigned to the available MAC-8 opcodes (hexadecimal 00 through FF or binary 00000000 through 11111111). The blank entries in the *Function* column identify the currently unassigned opcodes. If a range appears in the *Hex* column, the mode indicator (denoted by *xx* or *xxx* in the *Binary* column) is added to the lowest number in the range to determine the opcode for that function:

- *xx* is any monadic mode number (0 to 3)
- *xxx* is any dyadic mode number (0 to 7). However, mode 4 (memory-to-memory) can be used only for 8-bit instructions. For mode 4, the 8-bit instructions use the opcode for the corresponding 16-bit instructions. The opcodes for mode 4 in the 8-bit instruction group are not used.

The entry in the *Cond* column identifies the effect that function has on the Condition Register, but does not indicate whether the results are valid (refer to the instruction's description in Chapter 3 for a list of its testable conditions). These characters denote the following:

A	Alters negative, odd, zero, ones, overflow, and carry
F	Alters negative, odd, zero, ones
M	Mask determines conditions affected
N	No effect; conditions unchanged
S	Byte popped from stack determines conditions affected.

OPCODE ASSIGNMENTS

Hex	Function	Cond	Binary
00	Illegal opcode	N	00000000
01	Set Conditions	M	00000001
02			00000010
03	Clear Conditions	M	00000011
04	Pop A	N	00000100
05	Pop CR	S	00000101
06	Push A	N	00000110
07	Push CR	N	00000111
08			00001000
09			00001001
0A			00001010
0B			00001011
0C	Find Left One	F	00001100
0D	Load SP	N	00001101
0E	Count Ones	F	00001110
0F			00001111
10-13			000100*xx*
14-17			000101*xx*
18-1B			000110*xx*

Opcode Assignments

Hex	Function	Cond	Binary
1C-1F			000111*xx*
20-23	Zero	N	001000*xx*
24-27	Negate	A	001001*xx*
28-2B	Increment/Decrement	A	001010*xx*
2C-2F	Complement	F	001011*xx*
30-33	Shift Arithmetic	A	001100*xx*
34-37	Rotate 8	A	001101*xx*
38-3B	Shift Logical	A	001110*xx*
3C-3F	Rotate 9	A	001111*xx*
40	Branch 2 False	N	01000000
41	Branch 1 False	N	01000001
42	Bump 4	N	01000010
43	Bump 8	N	01000011
44	Pop B	N	01000100
45	Pop RP	N	01000101
46	Push B	N	01000110
47	Push RP	N	01000111
48	Branch 2 True	N	01001000
49	Branch 1 True	N	01001001
4A	Debump 4	N	01001010
4B	Debump 8	N	01001011
4C	Find Left One, Clear	F	01001100
4D	Load RP	N	01001101
4E			01001110
4F			01001111
50			01010000
51			01010001
52	Branch on Reg Bit False	N	01010010
53	Branch on Mem Bit False	N	01010011
54			01010100
55			01010101
56			01010110
57			01010111
58	Branch 2	N	01011000
59	Branch 1	N	01011001
5A	Branch on Reg Bit True	N	01011010
5B	Branch on Mem Bit True	N	01011011
5C			01011100
5D			01011101
5E			01011110
5F			01011111
60	Zero 16	N	01100000
61	Call False	N	01100001
62	Sign Extend	N	01100010
63			01100011
64	Sub Return True	N	01100100
65	Sub Return False	N	01100101
66	Sub Return	N	01100110
67	Int Return	S	01100111
68	Increment/Decrement 16	A	01101000
69	Call True	N	01101001
6A	Swap Bytes	N	01101010
6B			01101011
6C			01101100
6D	Load Register Address	N	01101101
6E			01101110

Hex	Function	Cond	Binary
6F	Load Address	N	01101111
70			01110000
71			01110001
72			01110010
73	Branch on Mem Bit False ⧺	N	01110011
74			01110100
75	Add 8 Bits Logical	N	01110101
76			01110110
77			01110111
78	Halt	N	01111000
79	Call	N	01111001
7A			01111010
7B	Branch on Mem Bit True ⧺	N	01111011
7C			01111100
7D	Add 8 Bits Arithmetic	N	01111101
7E			01111110
7F	NOP	N	01111111
80-83	Move (modes 0-3)	F	10000*xxx*
84			10000100
85-87	Move (modes 5-7)	F	10000*xxx*
88-8B	XOR (modes 0-3)	F	10001*xxx*
8C			10001100
8D-8F	XOR (modes 5-7)	F	10001*xxx*
90-93	OR (modes 0-3)	F	10010*xxx*
94			10010100
95-97	OR (modes 5-7)	F	10010*xxx*
98-9B	AND (modes 0-3)	F	10011*xxx*
9C			10011100
9D-9F	AND (modes 5-7)	F	10011*xxx*
A0-A3	Subtract (modes 0-3)	A	10100*xxx*
A4			10100100
A5-A7	Subtract (modes 5-7)	A	10100*xxx*
A8-AB	Add (modes 0-3)	A	10101*xxx*
AC			10101100
AD-AF	Add (modes 5-7)	A	10101*xxx*
B0-B3	Compare (modes 0-3)	A	10110*xxx*
B4			10110100
B5-B7	Compare (modes 5-7)	A	10110*xxx*
B8-BB	Test (modes 0-3)	F	10111*xxx*
BC			10111100
BD-BF	Test (modes 5-7)	F	10111*xxx*
C0-C3	Move 16 (modes 0-3)	F	11000xxx
C4	Move (mode 4)	F	11000100
C5-C7	Move 16 (modes 5-7)	F	11000*xxx*
C8-CB	XOR 16 (modes 0-3)	F	11001*xxx*
CC	XOR (mode 4)	F	11001100
CD-CF	XOR 16 (modes 5-7)	F	11001*xxx*
D0-D3	OR 16 (modes 0-3)	F	11010*xxx*
D4	OR (mode 4)	F	11010100
D5-D7	OR 16 (modes 5-7)	F	11010*xxx*
D8-DB	AND 16 (modes 0-3)	F	11011*xxx*
DC	AND (mode 4)	F	11011100
DD-DF	AND 16 (modes 5-7)	F	11011*xxx*

Opcode Assignments

Hex	Function	Cond	Binary
E0-E3	Subtract 16 (modes 5-7)	A	11100*xxx*
E4	Subtract (mode 4)	A	11100100
E5-E7	Subtract 16 (modes 5-7)	A	11100*xxx*
E8-EB	Add 16 (modes 0-3)	A	11101*xxx*
EC	Add (mode 4)	A	11101100
ED-EF	Add 16 (modes 0-3)	A	11101*xxx*
F0-F3	Compare 16 (modes 0-3)	A	11110*xxx*
F4	Compare (mode 4)	A	11110100
F5-F7	Compare 16 (modes 5-7)	A	11110*xxx*
F8			11111000
F9			11111001
FA			11111010
FB			11111011
FC	Test (mode 4)	F	11111100
FD			11111101
FE			11111110
FF	Illegal Opcode		11111111

6.3 Addressing Modes

These tables identify the modes of addressing operands in monadic and dyadic instructions. The columns indicate the addressing mode, hexadecimal code mode indicator, and register-number range for the d/s byte (destination/source operand qualifiers).

Monadic Instructions

The table identifies the addressing modes for these single-operand instructions. The 8-bit monadic instructions can use any of these addressing modes. The 16-bit monadic instructions can use only the Register mode (code 0).

The operand qualifier occupies the d field of the d/s byte; the s field is not used here for addressing.

MONADIC ADDRESSING MODES

Address Mode Destination	Code	Range d
Register	0	0-15
Indirect	1	0-14
Direct	1	15
Offset Memory	2	0-14
Offset Stack	2	15
Auto Increment	3	0-15

Dyadic Instructions

The table lists the addressing modes for the destination and source operands. However, mode 4 is *not* available for 16-bit dyadic instructions.

DYADIC ADDRESSING MODES				
Address Mode		Code	Range	
Destination	Source		d	s
Register	Register	0	0-15	0-14
Register	Immediate	0	0-15	15
Indirect	Register	1	0-14	0-14
Indirect	Immediate	1	0-14	15
Direct	Register	1	15	0-14
Direct	Immediate	1	15	15
Offset Memory	Register	2	0-14	0-14
Offset Memory	Immediate	2	0-14	15
Offset Stack	Register	2	15	0-14
Offset Stack	Immediate	2	15	15
Auto Increment	Register	3	0-15	0-15
Auto Increment	Immediate	3	0-15	15
Offset Memory	Offset Memory	4	0-14	0-14
Offset Memory	Offset Stack	4	0-14	15
Offset Stack	Offset Memory	4	15	0-14
Offset Stack	Offset Stack	4	15	15
Register	Indirect	5	0-15	0-14
Register	Direct	5	0-15	15
Register	Offset Memory	6	0-15	0-14
Register	Offset Stack	6	0-15	15
Register	Auto Increment	7	0-15	0-15

6.4 Instruction Length

These tables summarize the length of the A/L instructions[1] in memory bytes and machine cycles. The addressing mode, rather than the operation to be performed, determines the instruction length for this group. The tables illustrate the instructions written in assembler language using the following notation:

op	denotes any valid operator
ad, as	denotes destination (ad) or source (as) A Register
bd, bs	denotes destination (bd) or source (bs) B Register
N, M	denotes 8-bit *byte* contained in instruction; accessed through Program Counter
W, V	denotes 16-bit *word* or address contained in instruction; accessed through Program Counter
*	indicates indirection; location following the * points to operand

---

1. Refer to the descriptions in Chapter 3 for the instruction length of all other instructions.

Instruction Length

Monadic Instructions

These instructions involve a single operand.

MONADIC INSTRUCTIONS

Instruction	Mode	Range		Bytes	Cycles
		d	s		
8-bit Instructions					
ad = ad _op_	0	0-15	-	2	7
*bd = *bd _op_	1	0-14	-	2	11
*W = *W _op_	1	15	-	4	12
*[bd+N] = *[bd+N] _op_	2	0-14	-	3	12
*[sp+N] = *[sp+N] _op_	2	15	-	3	12
*bd++ = *bd++ _op_	3	0-15	-	2	14
16-bit Instructions					
bd = bd _op_	0	0-15	-	2	12

Dyadic Instructions

These instructions have both a source and a destination operand.

DYADIC 8-BIT INSTRUCTIONS

Instruction	Mode	Range		Bytes	Cycles
		d	s		
ad = ad _op_ as	0	0-15	0-14	2	9
ad = ad _op_ N	0	0-15	15	3	10
*bd = *bd _op_ as	1	0-14	0-14	2	12
*bd = *bd _op_ N	1	0-14	15	3	13
*W = *W _op_ as	1	15	0-14	4	13
*W = *W _op_ N	1	15	15	5	14
*[bd+N] = *[bd+N] _op_ as	2	0-14	0-14	3	13
*[bd+N] = *[bd+N] _op_ M	2	0-14	15	4	14
*[sp+N] = *[sp+N] _op_ as	2	15	0-14	3	13
*[sp+N]= *[sp+N] _op_ M	2	15	15	4	14
*bd++ = *bd++ _op_ as	3	0-15	0-14	2	15
*bd++ = *bd++ _op_ N	3	0-15	15	3	16
*[bd+N] = *[bd+N] _op_ *[bs+M]	4	0-14	0-14	4	22
*[bd+N] = *[bd+N] _op_ *[sp+M]	4	0-14	15	4	22
*[sp+N] = *[sp+N] _op_ *[bs+M]	4	15	0-14	4	22
*[sp+N] = *[sp+N] _op_ *[sp+M]	4	15	15	4	22
ad = ad _op_ *bs	5	0-15	0-14	2	12
ad = ad _op_ *W	5	0-15	15	4	13
ad = ad _op_ *[bs+N]	6	0-15	0-14	3	13
ad = ad _op_ *[sp+N]	6	0-15	15	3	13
ad = ad _op_ *bs++	7	0-15	0-15	2	15

DYADIC 16-BIT INSTRUCTIONS

Instruction	Mode	Range		Bytes	Cycles
		d	s		
bd = bd *op* bs	0	0-15	0-14	2	14
bd = bd *op* W	0	0-15	15	4	16
*bd = *bd *op* bs	1	0-14	0-14	2	17
*bd = *bd *op* W	1	0-14	15	4	19
*W = *W *op* bs	1	15	0-14	4	18
*W = *W *op* V	1	15	15	6	20
*[bd+N] = *[bd+N] *op* bs	2	0-14	0-14	3	18
*[bd+N] = *[bd+N] *op* W	2	0-14	15	5	20
*[sp+N] = *[sp+N] *op* bs	2	15	0-14	3	18
*[sp+N] = *[sp+N] *op* W	2	15	15	5	20
*bd++ = *bd++ *op* bs	3	0-15	0-14	2	20
*bd++ = *bd++ *op* W	3	0-15	15	4	22
bd = bd *op* *bs	5	0-15	0-14	2	17
bd = bd *op* *W	5	0-15	15	4	18
bd = bd *op* *[bs+W]	6	0-15	0-14	3	18
bd = bd *op* *[sp+W]	6	0-15	15	3	18
bd = bd *op* *bs++	7	0-15	0-15	2	20

Bell Laboratories

MAC-8 PROCESSOR USER'S MANUAL